AI VOID FIELD GUIDE

Mastering Modern Systems Engineering: From Apps to Architectures

Learn how small applications evolve into large-scale architectures using timeless engineering principles, covering distributed systems, scalability, resilience, and AI integration.

AI VOID

2026-05-15 21:16 IST

Contents

CHAPTER 01

From Monolith to Microservices: The Why and How of Distributed Systems

Imagine your application as a small sapling. It's easy to plant, easy to water, and grows quickly. But what happens when that sapling needs to become a towering tree, supporting a bustling ecosystem of users and complex features? This is the journey we'll embark on – understanding how software systems evolve from simple, unified structures to complex, distributed architectures.

In this chapter, we'll explore the fundamental shift from monolithic applications to distributed systems, often exemplified by microservices. We'll uncover the 'why' behind this evolution, examining the challenges that push systems towards distribution, and begin to understand the 'how' by looking at the core principles that guide this transformation. This isn't just about technology; it's about a mindset for building scalable, resilient, and manageable systems that can stand the test of time and support even the most sophisticated AI agents.

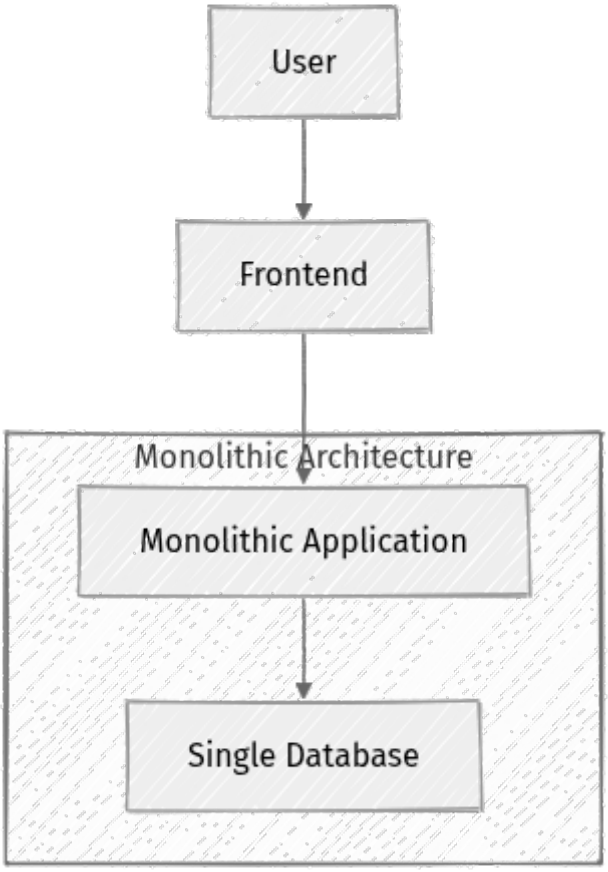
This chapter is your starting point, requiring no prior knowledge of distributed systems. We'll build our understanding step-by-step, ensuring you grasp the foundational concepts before we dive into the intricate details in subsequent chapters.

The Humble Monolith: A Unified Beginning

Most applications begin life as a monolith. Think of a monolith as a single, self-contained unit where all functional components – user interface, business logic, and data access layers – are tightly coupled and run as a single process. It's a pragmatic choice for getting started quickly.

What is a Monolith?

A monolithic application bundles all its functionalities into one deployable unit. For example, in an e-commerce platform, the user authentication, product catalog, shopping cart, and payment processing might all reside within the same codebase and run on a single server instance. All components share the same process and often, a single database.



- **User:** Interacts with the frontend.
- **Frontend:** The user interface (web or mobile).
- **Monolithic Application:** A single application process containing all business logic.
- **Single Database:** All data for the entire application is stored here.

Why Start with a Monolith?

Monoliths offer several compelling advantages, especially in the early stages of a project:

- **Simplicity in Development:** With everything in one place, development is straightforward. You don't need to worry about inter-service communication or distributed transactions.
- **Easier Debugging:** Tracing issues is often simpler as you can follow the execution path within a single process and usually one language.
- **Simple Deployment:** You only need to build and deploy one artifact. This makes continuous integration and continuous delivery (CI/CD) pipelines initially less complex.

- **Lower Operational Overhead (Initially):** Managing one application instance is typically easier than managing many.

⚡ `Real-world insight:` Many highly successful startups, including early versions of giants like Amazon, eBay, and Netflix, began with monolithic architectures. It's a pragmatic choice for proving a concept quickly and gaining market traction.

Growing Pains: When Monoliths Suffer

While great for starting, monoliths eventually face significant challenges as applications grow in size, complexity, and user base. These "growing pains" often become the driving force behind considering a distributed architecture.

Common Monolithic Challenges

- **Scalability Limits:** To scale a monolithic application, you typically have to scale the entire application (e.g., run more copies of the whole thing). If only one small part (like image processing) is a bottleneck, you still have to scale the entire, potentially resource-heavy application. This is inefficient and costly.
- **Deployment Bottlenecks:** Even a tiny change requires redeploying the entire application. This can lead to longer deployment times, increased risk of introducing new bugs, and a slower pace of innovation.
- **Technology Lock-in:** The entire application usually uses a single technology stack (e.g., Java with Spring Boot, Python with Django). It's difficult to introduce new languages or frameworks for specific features that might be better suited for them.
- **Team Friction and Slow Development:** As the codebase grows, it becomes harder for large teams to work on it simultaneously without stepping on each other's toes. Merging code can become a nightmare, leading to slower delivery.
- **Resilience Issues:** A failure in one component of the monolith can bring down the entire application. There's no isolation between parts, making the system brittle.
- **Increased Complexity:** Over time, a monolith can become a "big ball of mud," where dependencies are tangled, and understanding the system as a whole becomes incredibly difficult.

⚠️ `What can go wrong:` Imagine an e-commerce monolith where a sudden surge in traffic to the "recommendation engine" component causes the entire payment system to crash because they share resources. This lack of isolation is a critical vulnerability for production systems.

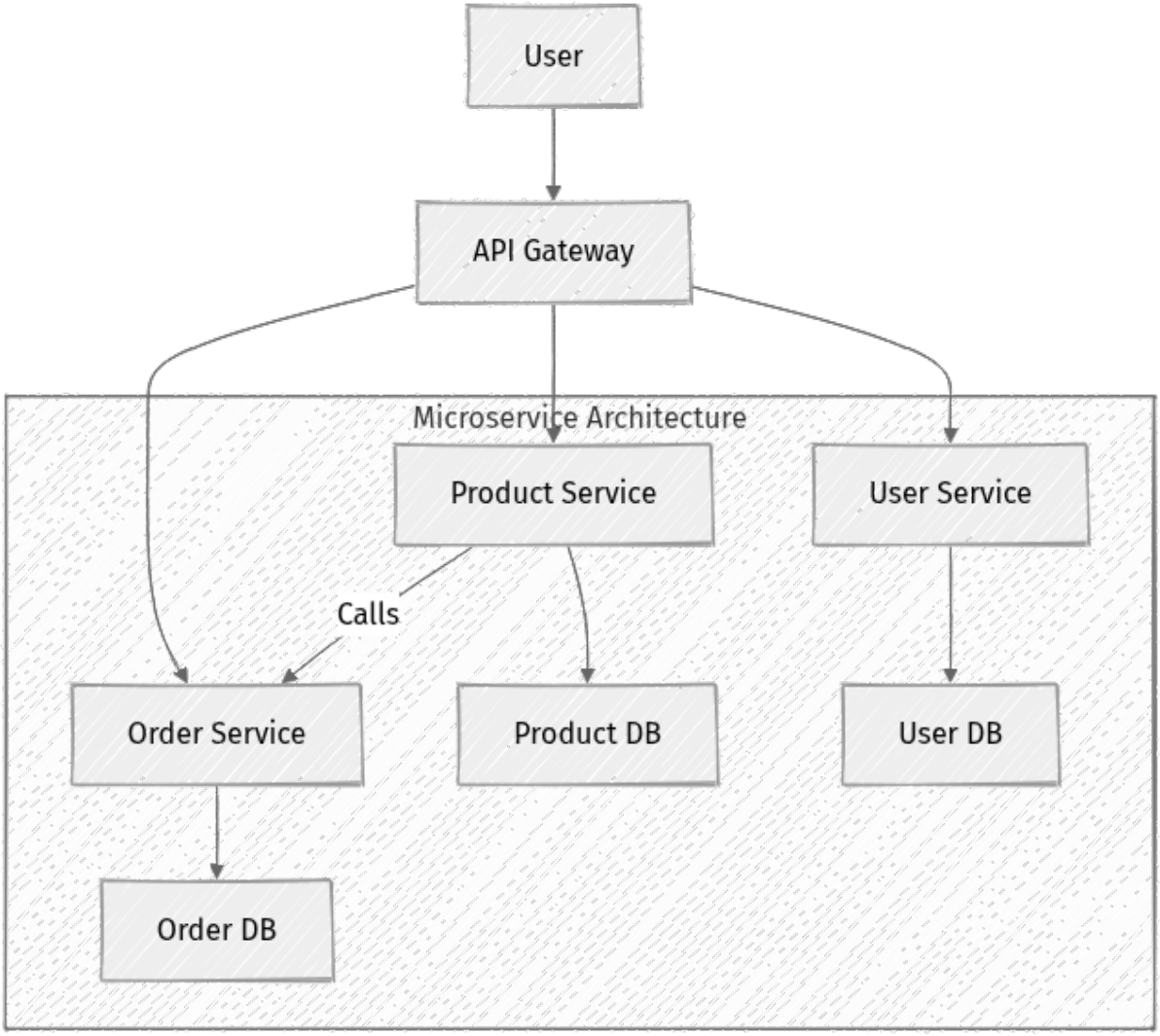
Embracing Distribution: The Microservices Approach

To address the limitations of monoliths, engineers often turn to distributed systems, with microservices being a prominent architectural style within this paradigm.

What are Distributed Systems and Microservices?

A **distributed system** is a collection of independent computing elements that appear to its users as a single coherent system. These elements communicate over a network to achieve a common goal.

Microservices are an architectural approach where an application is composed of small, independent services that communicate over well-defined APIs. Each service typically owns its data, can be developed by a small team, and deployed independently. They are a specific style of distributed system.



- **User:** Interacts with the frontend.
- **API Gateway:** A single entry point that routes requests to appropriate services.
- **Product Service, User Service, Order Service:** Independent microservices, each responsible for a specific business capability.
- **Product DB, User DB, Order DB:** Each service owns its dedicated data store.
- **Calls:** Services communicate with each other over the network via APIs.

Why Microservices? The Benefits

Microservices directly tackle the problems faced by monoliths, offering several compelling advantages:

- **Independent Scalability:** You can scale individual services based on their specific demand. If the "Product Catalog" service needs more resources, you only scale that service, not the entire application. This is much more efficient and cost-effective.
- **Independent Deployment:** Each service can be deployed, updated, and rolled back independently. This accelerates development cycles, reduces risk, and allows for continuous delivery of small changes.
- **Technology Heterogeneity:** Different services can be built using different programming languages, frameworks, and data stores, allowing teams to choose the best tool for the job. This prevents technology lock-in.
- **Improved Resilience:** If one service fails, it doesn't necessarily bring down the entire application. Other services can continue to function, leading to a more robust system.
- **Team Autonomy:** Small, cross-functional teams can own, develop, and operate specific services end-to-end, fostering greater ownership and faster decision-making.
- **Easier Code Management:** Smaller, focused codebases are easier to understand, maintain, and refactor.

`📌 Key Idea:` Microservices are a strategy for managing complexity and enabling agility, not a silver bullet. The goal is to maximize the benefits of independent development and deployment while managing the inherent complexities of distributed systems.

The Trade-Offs: When Not to Use Microservices

While powerful, microservices introduce their own set of complexities. It's crucial to understand these trade-offs to avoid over-engineering.

The Hidden Costs of Distribution

- **Increased Operational Complexity:** Managing many independent services, each with its own deployment, monitoring, and logging, is significantly more complex than managing a single monolith. You'll need more sophisticated infrastructure and tooling.

- **Distributed Debugging:** Tracing requests across multiple services, potentially written in different languages, can be challenging. You need specialized tools for distributed tracing.
- **Data Consistency Challenges:** Maintaining data consistency across multiple, independent databases is a complex problem that requires careful design (e.g., eventual consistency, distributed transactions).
- **Network Latency and Reliability:** Services communicate over a network, which introduces latency and the possibility of network failures. This needs to be explicitly handled in every service.
- **Overhead of Inter-service Communication:** Every call between services incurs network overhead, which can impact performance if not designed carefully.
- **Cost:** While efficient at scale, the infrastructure and tools required to manage a robust microservices architecture can be more expensive than a simple monolithic setup, especially in the early stages.

🧠 `Important:` Don't build a distributed system unless you have a compelling reason to. The overhead is substantial. Starting with a well-architected monolith and selectively extracting services as needed (often called the "strangler fig pattern") is often a safer approach. This allows you to gradually migrate functionality without a risky big-bang rewrite.

Timeless Principles of Distributed Systems Thinking

Regardless of whether you're building a traditional application or an AI agent workflow, certain engineering principles are paramount when dealing with distributed systems. These are timeless because they address the fundamental challenges of coordinating independent components.

1. **Decomposition:** Break down a large problem into smaller, manageable pieces, typically based on business capabilities or domains. For an AI agent, this might mean separating a "Perception Service" from a "Planning Service" and an "Action Execution Service."

2. **Loose Coupling & High Cohesion:**
    - **Loose Coupling:** Services should know as little as possible about the internal workings of other services. They interact via well-defined APIs, minimizing dependencies.
    - **High Cohesion:** The code within a single service should be highly related and focused on a single responsibility. This makes the service easier to understand and maintain.
3. **Independent Deployment:** Each service should be deployable without requiring changes or redeployments of other services. This is key to agility.
4. **Data Ownership:** Each service should own its data store, ensuring autonomy and preventing direct database access from other services. This enforces loose coupling and allows services to evolve their data schema independently.
5. **Resilience:** Design services to withstand failures. This involves techniques like retries, circuit breakers, and timeouts (which we'll explore in future chapters). Assume failures will happen.
6. **Observability:** Understand what's happening inside your distributed system. This includes comprehensive logging, metrics, and distributed tracing. Without it, debugging becomes nearly impossible in a system with many moving parts.
7. **Automation:** Automate everything from deployment to monitoring to scaling. Manual processes don't scale with distributed systems and introduce human error.

`⚡ Real-world insight:` Even complex AI agentic systems, which might involve multiple specialized agents collaborating, rely on these principles. Each agent or sub-agent can be thought of as a service, requiring robust communication, independent logic, and clear interfaces to orchestrate complex tasks. For example, a "research agent" might be a separate service from a "code generation agent," communicating through a central orchestrator.

Practical Application: Decomposing an AI Agent Workflow

While this chapter is conceptual, the best way to internalize these principles is to apply them. Since we're not writing code yet, our "implementation" will be a design exercise.

Challenge: Imagine you're building an advanced AI agent that can research a topic, generate code based on that research, and then deploy the code to a test environment. Currently, this is all handled by one massive script. How would you start thinking about decomposing this into potential microservices?

Hint: Think about the distinct, independent capabilities or domains within this workflow. What are the natural boundaries where you could draw a line and say, "This part does one thing, and it does it well"? Consider the principles of high cohesion and loose coupling.

What to Observe/Learn: This exercise helps you practice identifying potential service boundaries and applying the principle of decomposition. You'll start to see how different responsibilities can be isolated, laying the groundwork for designing truly distributed systems. Take a moment to sketch out your ideas on paper or in a simple text editor.

Common Pitfalls & Troubleshooting in Early Stages

As you begin to think about distributed systems, be aware of these common traps. Avoiding them early can save immense effort later.

1. **Premature Microservices:** Don't start with microservices unless you truly understand the problem they solve for your specific context. Many applications thrive as monoliths for a long time. Over-engineering too early can kill a project by adding unnecessary complexity and overhead before the core problem is even validated.
2. **Distributed Monoliths:** This is a common anti-pattern where you split a monolith into multiple services, but they remain tightly coupled, share a single database, or are deployed together. You end up with all the complexity of distributed systems with none of the benefits of true independence.
3. **Ignoring Network Problems:** Forgetting that network calls can fail, be slow, or drop messages is a recipe for disaster in distributed systems. Always assume the network is unreliable and design your services to handle partial failures gracefully. This includes implementing timeouts, retries, and fallback mechanisms.

Summary: The Journey Ahead

In this chapter, we've laid the groundwork for understanding the evolution of software architectures:

- **Monoliths** offer initial simplicity and faster development but face limitations in scalability, deployment, and team agility as systems grow.
- **Microservices** address these challenges by breaking applications into small, independent, and communicative services.
- This architectural shift introduces **new complexities** related to operations, debugging, and data consistency, requiring careful management.
- **Timeless engineering principles** like decomposition, loose coupling, resilience, and observability are crucial for success in distributed environments, whether for traditional applications or advanced AI agent systems.
- Crucially, always consider the **trade-offs** and avoid premature optimization or blindly applying patterns.

The journey from a monolith to a robust distributed system is not trivial, but with a solid understanding of these foundational concepts and principles, you're well-equipped to navigate it. In our next chapter, we'll begin to explore the very first steps of building distributed systems by diving into how services communicate with each other.

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

References

- Microservices Architecture Style - Azure Architecture Center
- Monolith First (Martin Fowler)
- Domain-Driven Design (Domain Language)

CHAPTER 02

Scaling with Reverse Proxies and API Gateways

Imagine your application starts small, a single server humming along, directly serving every user request. What happens when users multiply by thousands, or even millions? Direct access quickly becomes a bottleneck, a security risk, and a nightmare to manage. This is where reverse proxies and API gateways step in, transforming a fragile single point into a robust, scalable entry for your entire system.

In this chapter, we'll peel back the layers of how modern systems handle inbound traffic, learning the timeless engineering principles behind reverse proxies and API gateways. You'll understand not just what these components are, but why they are indispensable for building scalable, resilient, and secure architectures, especially in the context of distributed systems and emerging AI agent workflows. We'll explore their core functionalities, their evolution, and how to think about integrating them into your designs without falling into the trap of over-engineering.

The Foundation: Understanding the Reverse Proxy

Before we dive into the complexities of large-scale systems, let's establish a fundamental building block: the reverse proxy.

What is a Reverse Proxy?

At its core, a **reverse proxy** is a server that sits in front of one or more web servers and forwards client requests to them. Instead of clients communicating directly with your application servers, they communicate with the reverse proxy. The proxy then decides which backend server should handle the request, fetches the response, and sends it back to the client. The client never knows which specific server actually processed its request.

Why does it exist? Imagine a busy restaurant. Instead of every customer walking into the kitchen to place an order directly with a chef, there's a host or maitre d'. The host takes your order, directs it to the right station (grill, salad, pastry), and brings you the finished meal. The host is the reverse proxy. It exists to manage inbound traffic, distribute work, and present a single, consistent interface to the outside world.

Key Benefits of a Reverse Proxy

Reverse proxies aren't just about hiding your backend servers; they offer a suite of critical benefits that are essential for any production-grade application.

1. Load Balancing

What it is: Distributing incoming network traffic across multiple backend servers to ensure no single server is overwhelmed. **Why it matters:** As user traffic grows, you'll need more than one application server. A reverse proxy intelligently sends requests to the server that's least busy or most available, ensuring optimal resource utilization and preventing bottlenecks. This is crucial for horizontal scaling. **How it works:** Reverse proxies use various algorithms (like round-robin, least connections, or IP hash) to decide which backend server receives the next request.

2. SSL/TLS Termination

What it is: Handling the encryption and decryption of secure (HTTPS) connections. **Why it matters:** Establishing and maintaining SSL/TLS connections is computationally intensive. By offloading this task to the reverse proxy, your backend application servers can focus solely on processing business logic, significantly improving their performance. **How it works:** The reverse proxy holds the SSL certificate and manages the secure connection with the client. It then communicates with the backend servers, often over unencrypted (but internal and secure) HTTP, reducing overhead on the application servers.

3. Caching Static Content

What it is: Storing frequently accessed static files (images, CSS, JavaScript) directly at the proxy layer. **Why it matters:** If a user requests an image, and the reverse proxy has a cached copy, it can serve that copy directly without bothering the backend server. This significantly reduces latency for the client, reduces load on backend servers, and saves bandwidth. **How it works:** The proxy checks its cache first. If the content is there and fresh, it returns it immediately. Otherwise, it forwards the request to the backend, caches the response, and then returns it to the client.

4. Security Enhancements

What it is: Acting as the first line of defense against various network attacks. **Why it matters:** By presenting a single public endpoint, the reverse proxy can filter malicious traffic, block known attack patterns (like SQL injection or cross-site scripting via a Web Application Firewall

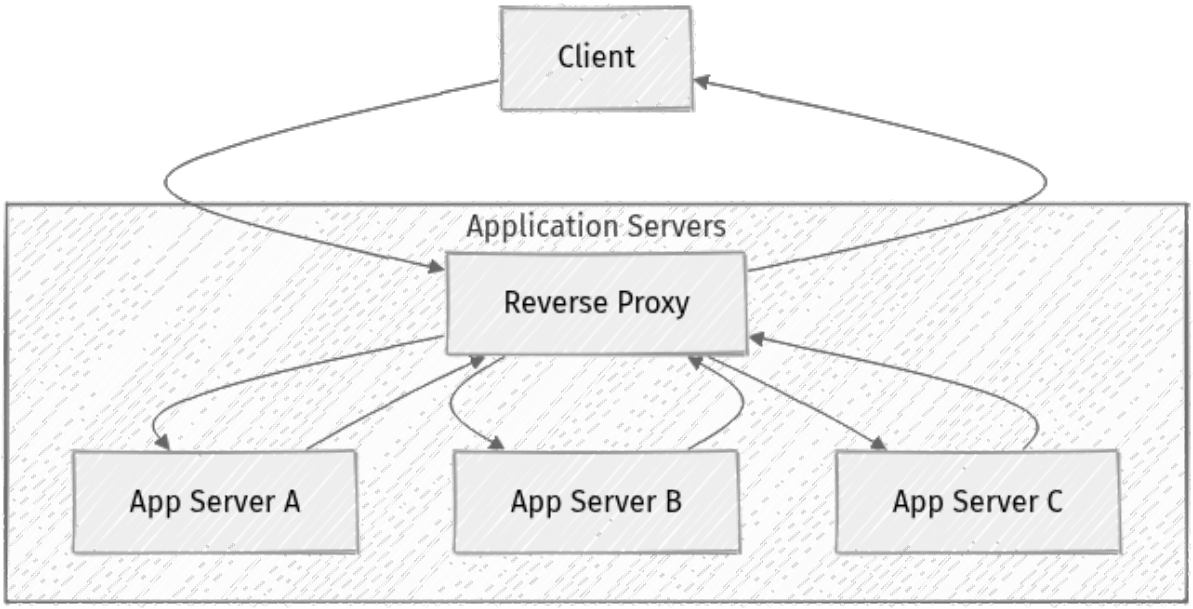
- WAF), and hide the internal network topology from attackers. **How it works:** It inspects incoming requests, applying security rules and potentially integrating with security services before forwarding them to internal servers.

5. Compression and Optimization

What it is: Compressing responses before sending them to the client to reduce data transfer size. **Why it matters:** Smaller responses mean faster load times for users, especially on slower networks, and reduced bandwidth costs for you. **How it works:** The proxy compresses text-based responses (HTML, CSS, JSON) using algorithms like Gzip or Brotli before sending them out, and decompresses incoming requests if necessary.

Visualizing the Reverse Proxy Flow

Let's illustrate the basic flow of a client request through a reverse proxy to multiple backend servers.



In this diagram, the `Client` sends a request to the `Reverse Proxy`. The `Reverse Proxy` then forwards that request to one of the `Application Servers` (A, B, or C) based on its load balancing strategy. The chosen `Application Server` processes the request and sends the response back through the `Reverse Proxy` to the `Client`.

Evolving to API Gateways: The Microservices Era

As applications grow and adopt microservices architectures, the simple reverse proxy often isn't enough. We need more intelligent routing, security, and management at the edge. This is where the **API Gateway** comes in.

What is an API Gateway?

An API Gateway is an evolution of the reverse proxy, specifically designed for microservices architectures. While a reverse proxy primarily handles traffic distribution and basic optimizations, an API Gateway adds a layer of intelligence, acting as a single entry point for all API requests. It encapsulates the internal system architecture and provides a tailored API to each client.

Why does it exist? In a microservices world, you might have dozens or hundreds of small services, each with its own API. A mobile app might need data from 5 different services to render a single screen. Without an API Gateway, the client would have to make 5 separate requests, manage authentication for each, and combine the results. This is complex, inefficient, and tightly couples the client to the internal service structure.

The API Gateway solves this by:

- **Aggregating requests:** A single request to the gateway can trigger multiple internal service calls, simplifying client-side logic.
- **Decoupling clients from services:** Clients only know about the gateway, not the individual services, allowing internal changes without impacting external consumers.
- **Centralizing cross-cutting concerns:** Authentication, authorization, rate limiting, logging, and monitoring are handled once at the gateway, rather than redundantly in every service.

Advanced Features of an API Gateway

API Gateways extend the capabilities of reverse proxies with features crucial for distributed systems.

1. Authentication and Authorization

What it is: Verifying the identity of the client and ensuring they have permission to access the requested resources. **Why it matters:** Centralizing security at the edge means individual backend services don't need to implement their own authentication logic. The gateway can validate tokens (e.g., JWTs) and pass user context to downstream services, simplifying service development and reducing security surface area. **How it works:** The gateway intercepts requests, extracts credentials (e.g., API keys, OAuth tokens), validates them with an identity provider, and either allows or denies the request.

2. Rate Limiting

What it is: Controlling the number of requests a client can make to your APIs within a given timeframe. **Why it matters:** Prevents abuse, protects backend services from being overwhelmed by traffic spikes, and ensures fair usage among clients. This is especially critical for resource-intensive AI agent services. **How it works:** The gateway tracks requests per client (e.g., by IP address or API key) and blocks requests that exceed predefined thresholds.

3. Request/Response Transformation

What it is: Modifying the structure or content of requests before forwarding them to a service, or responses before sending them back to the client. **Why it matters:** Allows clients to interact with a consistent API schema even if backend services have different versions or data formats. It can also strip sensitive information from responses before they leave your system. **How it works:** The gateway applies rules (e.g., JSON schema transformations, header modifications, data masking) to incoming and outgoing data.

4. Intelligent Routing and Service Discovery

What it is: Dynamically directing requests to the correct backend service instance, even as services scale up and down. **Why it matters:** In a microservices architecture, service instances are constantly starting, stopping, and scaling. The gateway needs to know where to find the currently active and healthy services without manual configuration. **How it works:** The gateway integrates with a service discovery mechanism (e.g., Kubernetes, Consul, Eureka) to find healthy service instances and route requests accordingly.

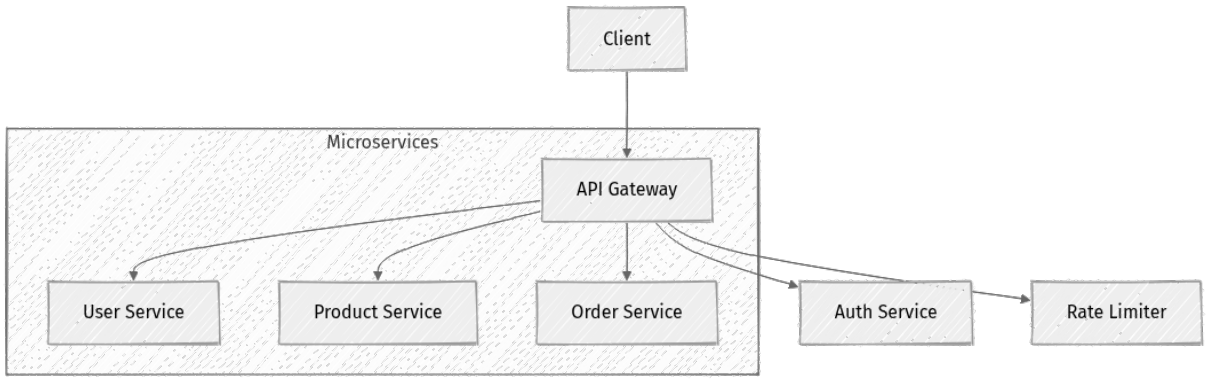
5. Circuit Breaking and Retries

What it is: Patterns to prevent cascading failures in distributed systems. **Why it matters:** If a backend service is unhealthy or slow, blindly sending more requests to it will only make things worse and can cause other services to fail. Circuit breakers stop traffic to failing services, and retries handle transient network

issues. **How it works:** The gateway monitors service health. If a service consistently fails, the circuit breaker "opens," preventing further requests for a period. Retries automatically re-send failed requests if the error is likely temporary (e.g., network glitch).

API Gateway in a Microservices World

Here's how an API Gateway fits into a microservices architecture:



In this setup, the `Client` sends a single request to the `API Gateway`. The `API Gateway` first interacts with `Auth Service` and `Rate Limiter` for security and traffic control. Then, based on the request, it routes to one or more `Microservices` (User, Product, Order), potentially aggregating responses before sending a single, unified response back to the `Client`.

When to Use Which: Reverse Proxy vs. API Gateway

It's important to understand the nuance and avoid over-engineering:

- **Use a Reverse Proxy when:**
  - You have a monolithic application or a small number of services.
  - Your primary needs are load balancing, SSL termination, static content caching, and basic security.
  - You want a simple, high-performance HTTP/TCP proxy.
  - Example tools: Nginx (current stable version 1.25.x as of 2026-05-15), HAProxy (current stable version 2.9.x as of 2026-05-15).

- **Use an API Gateway when:**
    - You have a microservices architecture with many services.
    - You need advanced features like authentication, authorization, rate limiting, request/response transformation, and sophisticated routing.
    - You want to provide a consistent, versioned API façade to external clients.
    - Example tools: Kong Gateway (current stable version 3.6.x as of 2026-05-15), AWS API Gateway, Azure API Management, Google Cloud Apigee.

🧠 **Important:** Don't reach for an API Gateway if a simple reverse proxy suffices. Over-engineering with a full-blown API Gateway for a small application introduces unnecessary complexity and operational overhead. Start simple and evolve as your needs dictate.

Practical Application: Conceptual API Gateway Design

Since this guide focuses on timeless principles rather than specific vendor tools, let's think about the decisions involved in setting up an API Gateway for a hypothetical AI Agent Orchestration platform. This section will guide you through designing the configuration, step by step.

Imagine you're building a system where various AI agents (e.g., a "Research Agent," a "Code Generation Agent," a "Translation Agent") expose APIs. A central "Orchestration Agent" needs to call them, and external users need to interact with the orchestration layer.

Step 1: Define Your Entry Point and Basic Routing

First, decide on the public URL for your gateway. Then, establish basic routes that direct incoming requests to your core services.

```
# Conceptual API Gateway Configuration - Core Routing
# This is NOT runnable code, but a conceptual representation for design
thinking.

# Define the public endpoint for your entire platform
public_domain: api.myagentplatform.com

# Map incoming URL paths to internal services
routes:
  - path: /agents/research/*
    target_service: research-agent-service
```

```
    strip_prefix: /agents/research
    description: Routes requests to the AI Research Agent
  - path: /agents/code/*
    target_service: code-gen-agent-service
    strip_prefix: /agents/code
    description: Routes requests to the AI Code Generation Agent
  - path: /orchestrator/*
    target_service: orchestration-service
    strip_prefix: /orchestrator
    description: Routes requests to the main Orchestration Agent
  - path: /auth/*
    target_service: auth-service
    strip_prefix: /auth
    description: Routes authentication requests to the Authentication Service
```

Explanation:

- `public_domain`: This is the public face of your API. All external requests come here first.
- `routes`: This section defines how specific incoming URL paths map to your internal services.
- `path`: The URL path segment the gateway listens for (e.g., `/agents/research/query`). The `*` acts as a wildcard.
- `target_service`: The internal logical name of the service to which the request should be forwarded. This typically maps to a service discovery entry.
- `strip_prefix`: Removes the matched path segment (e.g., `/agents/research`) from the URL before forwarding to the target service. This keeps the internal service's API cleaner, as it doesn't need to know its public prefix.

Step 2: Implement Cross-Cutting Concerns

Now, let's add common features like authentication, authorization, and rate limiting. These policies are applied before routing to any specific service, ensuring consistent security and traffic management.

```
# Conceptual API Gateway Configuration - Global Policies

# Global Policies applied to all incoming requests by default
policies:
  - name: authentication
    type: JWT_Validation
    jwks_uri: https://auth.myagentplatform.com/.well-known/jwks.json
    required_claims:
      - sub
      - role
    # Allow unauthenticated access to /auth/* paths (e.g., for login/signup)
    exclude_paths: ["/auth/*"]
```

```
    description: Validates JWT tokens for all API requests.

  - name: authorization
    type: RBAC
    policy_engine_endpoint: http://internal-policy-service/authorize
    description: Checks user roles against resource permissions using an
internal policy service.

  - name: rate_limiting
    type: FixedWindow
    rate: 100 # requests per minute
    per: user_id # Apply this limit per authenticated user
    description: Limits requests to 100 per minute per user globally.
```

Explanation:

- `policies`: This section defines global rules that apply to most or all requests.
- `authentication`: Specifies that JWT tokens should be validated using a public key set (`jwks_uri`). It also lists required claims (e.g., `sub` for subject, `role` for user role) and importantly, excludes the `/auth/*` path, as authentication requests themselves shouldn't require prior authentication.
- `authorization`: Implements Role-Based Access Control (RBAC) by querying an internal `policy_engine_endpoint`. This service would determine if the authenticated user has permission for the requested action.
- `rate_limiting`: Sets a limit of 100 requests per minute, enforced per `user_id` (extracted from the authenticated JWT).

Step 3: Consider Service-Specific Enhancements

Some services might need unique treatment. For instance, your "Research Agent" might have a very long-running request, requiring a longer timeout, or a stricter rate limit due to high resource consumption.

```
# Conceptual API Gateway Configuration - Route Overrides

# Override global policies or add specific features for individual routes
route_overrides:
  - path: /agents/research/query
    # Increase timeout for potentially long-running AI research queries
    timeout_ms: 60000 # 60 seconds
    # Apply a different, stricter rate limit for heavy research tasks
    policies:
      - name: rate_limiting
        type: FixedWindow
        rate: 10 # requests per minute
        per: user_id
        description: Stricter limit for intensive research queries (10/min).

  - path: /orchestrator/status
    # Cache responses for status checks to reduce backend load
```

```
    policies:
      - name: caching
        type: TTL
        ttl_seconds: 5 # Cache for 5 seconds
        description: Caches orchestration status responses to improve
performance.
```

Explanation:

- `route_overrides`: This section allows you to apply specific configurations to individual routes, overriding or supplementing global policies.
- `/agents/research/query`: This specific endpoint might have a longer `timeout_ms` because AI research tasks can take time. It also applies a stricter rate limit than the global one, reflecting its resource-intensive nature.
- `/orchestrator/status`: This endpoint benefits from `caching` with a short Time-To-Live (TTL), reducing load on the orchestration service for frequently requested status updates.

This conceptual configuration demonstrates how an API Gateway allows you to centralize control, apply policies consistently, and tailor behavior for specific needs across a distributed system, especially vital for managing diverse AI agent workloads.

Mini-Challenge: Designing for a New AI Agent

You've successfully launched your platform. Now, a new "Image Generation Agent" is being developed. It will expose an API at `/agents/image/generate`. This agent is very resource-intensive (e.g., uses GPUs heavily), and you want to ensure it's protected from abuse and managed carefully.

Challenge: Draft the conceptual API Gateway configuration entries required for the new "Image Generation Agent" (`image-gen-agent-service`).

- It should be accessible via `api.myagentplatform.com/agents/image/*`.
- It requires the standard JWT `authentication` and `authorization` checks (these are global policies, so you don't need to re-declare them unless you want to override).
- It should have a much stricter `rate_limiting` policy: only **5 requests per user per minute** due to high GPU costs.

- Responses from this agent can be large (e.g., generated images), so ensure `compression` is enabled (assume this is a general gateway feature, but explicitly mention it as a consideration).

Hint: Think about how you'd combine the global policies with route-specific overrides. Remember, you only need to specify what changes or is added for this new route.

💡 **CLICK FOR A POSSIBLE SOLUTION (TRY IT YOURSELF FIRST!)**

```
# Conceptual API Gateway Configuration (solution snippet for Image Agent)
```

Add the new route to the 'routes' section:

```
routes:


  - path: /agents/image/*
    target_service: image-gen-agent-service
    strip_prefix: /agents/image
    description: Routes requests to the AI Image Generation Agent
```

Apply specific overrides for the image generation endpoint within 'route_overrides':

```
route_overrides:
```

```
- path: /agents/image/generate
# The global authentication and authorization policies would apply by default.

# We only need to specify overrides or additions here.

policies:

  - name: rate_limiting
    type: FixedWindow
    rate: 5 # requests per minute
    per: user_id
    description: Stricter limit for resource-intensive image generation (5/min).


# Assuming 'enable_compression' is a specific flag your gateway supports for a route.

enable_compression: true
```

What to observe/learn: You should notice that you don't need to re-declare `authentication` or `authorization` for the new route if they are already defined as global policies. Route overrides are for modifying or adding to the default behavior. The `rate_limiting` override demonstrates how to apply a

stricter policy for a specific, resource-intensive endpoint. Explicitly considering `compression` for potentially large responses is also key, as it directly impacts user experience and bandwidth costs.

Common Pitfalls & Troubleshooting

Even with robust components like reverse proxies and API gateways, things can go wrong. Understanding common pitfalls helps in designing resilient systems.

1. Single Point of Failure (SPOF)

⚠️ **What can go wrong:** If your reverse proxy or API Gateway is deployed as a single instance, and it fails, your entire application becomes inaccessible. This is a critical vulnerability. **Troubleshooting:** Implement high availability. This means running multiple instances of your gateway behind a hardware or software load balancer (often another, simpler reverse proxy, or cloud-managed load balancers). ⚡ **Real-world insight:** Cloud providers offer managed load balancers (e.g., AWS ELB/ALB, Azure Load Balancer/Application Gateway, Google Cloud Load Balancing) that are inherently highly available and distribute traffic across multiple gateway instances, abstracting away the complexity of managing individual proxy servers.

2. Over-engineering and Premature Optimization

⚠️ **What can go wrong:** Implementing a full-featured API Gateway when a simple reverse proxy (or even no proxy) would suffice for your current scale. This adds unnecessary complexity, configuration overhead, and a steeper learning curve, ultimately slowing down development. **Troubleshooting:** Start simple. Choose the simplest solution that meets your current needs. As your application grows and requirements become more complex (e.g., microservices adoption, advanced security needs, integrating many AI agents), then gradually introduce more sophisticated tools. 📌 **Key Idea:** Complexity is a cost. Only incur it when the benefits clearly outweigh that cost.

Assuming 'enable_compression' is a specific flag your gateway supports for a route.

3. Performance Overhead

⚠️ **What can go wrong:** Each layer in your architecture adds latency. An API Gateway, while powerful, introduces an additional hop for every request. If not optimized, this can lead to unacceptable response times, especially for latency-sensitive applications or AI agents requiring fast responses. **Troubleshooting:**

- **Optimize gateway configuration:** Minimize unnecessary processing (e.g., complex transformations if not needed).
- **Efficient routing:** Ensure service discovery is fast and cached where possible.
- **Caching:** Leverage caching at the gateway for static or frequently accessed dynamic content.
- **Monitor performance:** Continuously measure latency introduced by the gateway using observability tools. If it becomes a bottleneck, investigate.

4. Complex Configuration and Management

⚠️ **What can go wrong:** As you add more routes, policies, and transformations, the gateway configuration can become a tangled mess, difficult to understand, manage, and debug. This can lead to errors, security vulnerabilities, and operational headaches. **Troubleshooting:**

- **Version control:** Treat gateway configuration as code and store it in version control (e.g., Git).
- **Automation:** Use infrastructure as code (IaC) tools (e.g., Terraform, Pulumi) to manage gateway deployments and updates.
- **Modularity:** Break down complex configurations into smaller, manageable pieces (if your gateway supports it).
- **Clear documentation:** Document your routing rules and policies thoroughly, explaining the why behind each configuration.

Summary

You've taken a significant step in understanding how modern, scalable applications handle inbound traffic. We covered:

- **Reverse Proxies** as the foundational layer, providing essential capabilities like load balancing, SSL termination, caching, and basic security. They are ideal for simpler setups or as a robust initial layer.

Assuming 'enable_compression' is a specific flag your gateway supports for a route.

- **API Gateways** as an intelligent evolution tailored for microservices and complex distributed systems, offering advanced features like centralized authentication, authorization, rate limiting, request/response transformation, and intelligent routing.
- **The critical distinction** between when to use a simple reverse proxy versus a full-featured API Gateway, emphasizing the importance of avoiding premature optimization and matching the tool to the problem.
- **Conceptual configuration** to illustrate how these components are designed in practice, particularly within the context of managing diverse AI agent workflows.
- **Common pitfalls** such as single points of failure, over-engineering, performance overhead, and configuration complexity, along with practical strategies to mitigate them and build more resilient systems.

Understanding these components is crucial for designing systems that are not only scalable and resilient but also secure and manageable. As you move forward, remember that the goal is to build robust systems, not just to apply patterns blindly. Think critically about your specific needs and the tradeoffs involved.

References

- Microservices Architecture Style - Azure Architecture Center
- Introduction to NGINX Reverse Proxy
- API Gateway - Martin Fowler
- OWASP API Security Top 10 (2023)
- What is a Load Balancer? - Cloudflare

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 03

Service-to-Service Communication: Synchronous vs. Asynchronous

Welcome back, aspiring systems architect! In the previous chapter, we explored how a reverse proxy acts as the intelligent front door to our services. Now, let's venture deeper into the heart of distributed systems: **how services talk to each other**. Just like people communicate in different ways – a quick chat versus sending a detailed email – services also have distinct communication styles. Choosing the right one is fundamental to building scalable, resilient, and performant applications, especially as we integrate advanced AI agent workflows.

This chapter will guide you through the two primary modes of service-to-service communication: **synchronous** and **asynchronous**. We'll break down what each means, how they work, their strengths and weaknesses, and most importantly, when to use which. By the end, you'll have a clearer understanding of the tradeoffs involved and how to make informed decisions for your system's architecture, including those powering modern AI agent workflows.

To get the most out of this chapter, a basic grasp of what a "service" or "microservice" is, along with an understanding of network requests (like HTTP), will be helpful.

The Dance of Services: Synchronous Communication

Imagine you're ordering food at a restaurant. You tell the waiter your order, and you wait right there until they bring your food. You can't start eating until they deliver it. This "request-and-wait" model is the essence of synchronous communication in distributed systems.

What is Synchronous Communication?

Synchronous communication is a direct, blocking interaction where a client service sends a request to a server service and then pauses its own operation, waiting for a response before it can continue. It's a "call-and-response" pattern.

How It Works: A Direct Conversation

When Service A needs information or an action from Service B, it sends a request directly to Service B. Service A then waits for Service B to process the request and send back a response. Only after receiving that response (or a timeout) can Service A proceed with its next step.

A common example of synchronous communication is using **HTTP-based APIs**, often following the REST architectural style.

In this flow:

- The User makes a request to the Frontend.
- The Frontend calls the API Gateway.
- The API Gateway routes the request to the Product Service (Service A).
- The Product Service immediately calls the Inventory Service (Service B) to check stock.
- The Product Service waits for the Inventory Service's response.
- Once Service B responds, Service A processes it and sends its own response back up the chain.

When to Choose Synchronous Communication

Synchronous communication is often the default choice due to its simplicity and immediate feedback.

- **Immediate Response Required**: When the client absolutely needs an immediate result to continue its workflow. For instance, a user login request needs to know right now if the credentials are valid.
- **Blocking Operations**: For operations that inherently block, such as retrieving data from a database before rendering a page.
- **Simple Workflows**: For straightforward interactions between two or a few services where dependencies are clear and direct.

Advantages of Synchronous Communication

- **Simplicity**: Easier to understand, implement, and debug. The flow is linear and predictable.
- **Immediate Feedback**: The caller knows immediately if the operation succeeded or failed.

- **Straightforward Error Handling**: Errors can be returned directly to the caller, simplifying retry logic or user notification.

⚠️ What can go wrong: Downsides and Pitfalls

While simple, synchronous communication introduces significant challenges in distributed systems:

- **Tight Coupling**: Services become highly dependent on each other. If Service B is slow or fails, Service A (and potentially the entire chain) will also slow down or fail. This is known as a **cascading failure**.
- **Latency**: Each hop in a synchronous call adds network latency. A chain of 5 synchronous calls, each taking 50ms, adds at least 250ms to the total response time, even before processing.
- **Scalability Bottlenecks**: If Service B is under heavy load, it can become a bottleneck for all services calling it, leading to resource exhaustion (e.g., connection pools) and reduced throughput across the system.
- **Resource Consumption**: The calling service must keep a thread or connection open while waiting for a response, consuming valuable resources.
- **Resilience Challenges**: Retries and circuit breakers (which we'll cover later) are necessary to mitigate failures, but they add complexity.

The Mailroom Approach: Asynchronous Communication

Now, imagine you're sending a physical letter. You drop it in the mailbox, and you don't wait for the recipient to read it and reply immediately. You carry on with your day, trusting that the letter will eventually reach its destination. This "fire-and-forget" or "publish-and-subscribe" model is the core idea behind asynchronous communication.

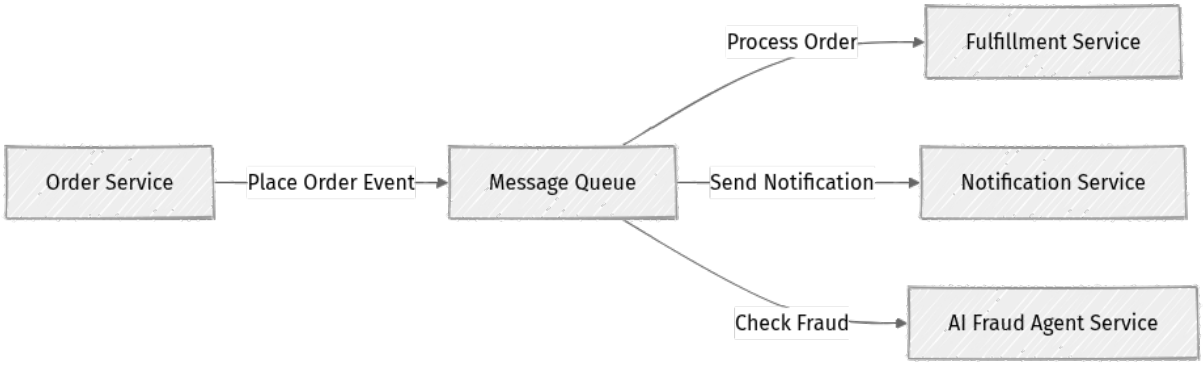
What is Asynchronous Communication?

Asynchronous communication is a non-blocking interaction where a client service sends a message or event and then continues its own operation without waiting for an immediate response. The message is typically placed into a mediator (like a message queue or event bus), which then delivers it to the appropriate server service.

How It Works: Messages and Events

When Service A needs Service B to perform an action, instead of calling Service B directly, it publishes a message or an event to a shared channel (e.g., a message queue). Service A then immediately proceeds with its next task. Service B, or another consumer, later retrieves the message from the channel and processes it.

Common technologies for asynchronous communication include **message queues** (like RabbitMQ, Apache Kafka, Amazon SQS) and **event streams**. These systems typically offer "at-least-once" delivery guarantees, ensuring messages aren't lost, and often support ordered processing within partitions.



In this flow:

- The Order Service (Service A) publishes a "Place Order" event to the Message Queue.
- Service A immediately continues, perhaps returning an "Order Received" status to the user.
- The Message Queue holds the event.
- The Fulfillment Service (Service B), Notification Service (Service C), and AI Fraud Agent Service pick up the event independently from the queue.
- Each service processes the event at its own pace, without blocking Service A.

When to Choose Asynchronous Communication

Asynchronous communication shines when dealing with complex, high-throughput, or long-running operations.

- **Decoupling Services**: When services need to interact without knowing too much about each other's existence or availability. This promotes independent deployment and scaling.

- **Long-Running Tasks**: For operations that take a significant amount of time (e.g., video encoding, complex data analysis, training an AI model). The user can be notified later.
- **High Throughput**: When a service needs to handle a large volume of requests without being constrained by the processing speed of downstream services.
- **Resilience**: If a downstream service is temporarily unavailable, messages can queue up and be processed once it recovers, preventing cascading failures.
- **Event-Driven Architectures**: For systems built around reacting to events, where multiple services might need to respond to a single occurrence.
- **AI Agent Workflows**: Many AI agent systems involve multi-step, potentially long-running tasks (e.g., an agent analyzing a large document, generating complex content, or orchestrating other agents). Asynchronous messaging allows agents to hand off tasks, continue processing, and pick up results when ready.

Advantages of Asynchronous Communication

- **Loose Coupling**: Services are independent. The producer doesn't need to know who the consumers are or if they're even online.
- **Increased Resilience**: Messages are durable in the queue. If a consumer fails, the message remains until it can be processed. This prevents upstream services from being blocked.
- **Improved Scalability**: Consumers can be scaled independently based on message load. Adding more consumers increases processing capacity without affecting the producer.
- **Enhanced Responsiveness**: The calling service doesn't wait for the operation to complete, allowing it to respond quickly to its own caller or continue other work.
- **Load Leveling**: Message queues act as buffers, smoothing out spikes in demand.

⚠️ What can go wrong: Downsides and Pitfalls

The power of asynchronous communication comes with added complexity.

- **Increased Complexity**: Introducing a message broker adds another component to manage, monitor, and secure. Debugging distributed asynchronous flows can be challenging.

- **Eventual Consistency**: Data might not be immediately consistent across all services. If Service A publishes an event and then immediately queries Service B, Service B might not have processed the event yet. This requires careful design.
- **Harder Error Handling and Tracing**: Tracing a request across multiple asynchronous hops (often involving different message IDs and correlation IDs) requires robust observability tools. Error handling needs to account for retries, dead-letter queues, and idempotent processing.
- **Message Ordering**: Guaranteeing the exact order of messages can be tricky with some queueing systems, especially under high load or with multiple consumers.
- **Over-engineering**: Applying asynchronous patterns unnecessarily for simple, low-volume interactions can introduce needless complexity and operational overhead. Always consider if the benefits outweigh the costs.

Designing a Communication Flow: A Step-by-Step Approach

Choosing between synchronous and asynchronous communication isn't always obvious. It's a fundamental design decision that impacts scalability, resilience, and operational complexity. Let's walk through a structured way to make this choice for any interaction in your system.

Step 1: Identify the User Experience and Immediate Feedback Needs

- **Question**: Does the user (human or another service) absolutely need an immediate response to proceed with their current task?
    - **Example**: A user trying to log in needs to know now if their credentials are valid. A user adding an item to a shopping cart expects immediate confirmation.
    - **Decision**: If yes, lean towards synchronous communication for this initial interaction. If no, and a delayed confirmation is acceptable, asynchronous is a strong candidate.

Step 2: Evaluate Task Duration and Complexity

- **Question**: How long does the operation typically take? Does it involve multiple steps, external calls, or heavy computation?
    - **Example**: Generating a complex report, training an AI model, processing a large video file, or orchestrating a multi-agent AI workflow are typically long-running. Retrieving a simple user profile from a database is usually fast.
    - **Decision**: For tasks that complete within tens to a few hundreds of milliseconds, synchronous might be fine. For anything longer, especially tasks that could take seconds, minutes, or even hours, asynchronous communication is almost always the better choice to prevent blocking and timeouts.

Step 3: Consider Fault Tolerance and Resilience Requirements

- **Question**: What happens if the downstream service is temporarily unavailable or slow? Can the upstream service gracefully handle this, or will it cause a cascading failure?
    - **Example**: If the payment processing service is down, should the entire order placement fail immediately, or can the order be accepted and payment retried later?
    - **Decision**: If the system must continue functioning even if a dependency is down, asynchronous communication provides resilience through message durability. If immediate failure is acceptable or desired (e.g., preventing a fraudulent transaction), synchronous might be chosen, but with robust retry and circuit breaker patterns.

Step 4: Assess Scalability and Decoupling Needs

- **Question**: Will this interaction experience high throughput? Do the services involved need to scale independently or be deployed separately?
    - **Example**: A notification service sending millions of emails per day needs to scale independently from the service generating those notifications.
    - **Decision**: High throughput and independent scaling are strong indicators for asynchronous communication. The message queue acts as a buffer and allows consumers to be scaled up or down without impacting the producer. If the interaction is low-volume and tightly coupled services are acceptable, synchronous might suffice.

Step 5: Diagram the Chosen Flow and Identify Integration Points

- Once you've made preliminary decisions, sketch out the interaction flow.
    - For synchronous interactions, identify direct API calls.
    - For asynchronous interactions, identify where messages are published to a queue/topic and where they are consumed.
- **Refinement**: Look for opportunities to convert synchronous calls to asynchronous ones where immediate feedback isn't critical, enhancing overall system resilience and performance.
- **Example Scenario**: An AI agent workflow might start with a synchronous API call to kick off a task, but then immediately transition to asynchronous messaging for all the long-running, multi-step sub-tasks the agent performs. The final result might be pushed to another queue for notification or stored for later retrieval via a synchronous poll.

Mini-Challenge: Design Communication for an Image Processing Workflow

Imagine you're building a system where users upload images, and an AI agent performs various transformations (e.g., resizing, applying filters, generating captions). The user should get an immediate "Upload successful, processing..." message, and then be notified when the image is fully processed and ready for download.

Your Challenge: Sketch out the communication flow between the following conceptual services, deciding which interactions should be synchronous and which asynchronous. Use a `flowchart LR` Mermaid diagram to illustrate your solution.

1. `Upload Service`: Receives raw images from users.
2. `AI Image Processor`: Applies transformations and generates captions.
3. `Notification Service`: Sends emails/in-app notifications to users.
4. `Storage Service`: Stores raw and processed images.

Hint: Think about what needs an immediate response versus what can happen in the background. Where might long-running tasks occur? How can you ensure the user gets timely updates without blocking their initial upload?

What to Observe/Learn: Consider how your choices impact the user experience, the resilience of the system, and the ability to scale different parts independently. Could a single slow image transformation block all other user uploads? How would you ensure the user gets notified reliably even if the notification service is temporarily down?

Common Pitfalls & Troubleshooting

Even with the best intentions, choosing and implementing communication patterns can lead to issues. Understanding these common traps is crucial for building robust systems.

Synchronous Pitfalls

- **Cascading Timeouts**: If one service in a synchronous chain is slow, it can cause timeouts in all upstream services, leading to widespread failures.
    - **Solution**: Implement aggressive timeouts (e.g., 100-200ms for internal service calls), circuit breakers (to stop sending requests to failing services), and retries with exponential backoff strategies.
- **Resource Exhaustion**: Keeping connections open while waiting for responses can exhaust connection pools or threads, leading to service unresponsiveness.
    - **Solution**: Use non-blocking I/O where possible (e.g., `async/await` in modern languages), carefully manage connection pools, and monitor resource usage (CPU, memory, open connections) to identify bottlenecks early.

Asynchronous Pitfalls

- **Eventual Consistency Headaches**: Data updates don't propagate instantly. If your application logic assumes immediate consistency, you'll encounter bugs.
    - **Solution**: Design your application to be tolerant of eventual consistency. Use techniques like idempotency (making operations repeatable without side effects) and "read-your-own-writes" consistency patterns, where a service might read from its own local cache immediately after writing, before the update propagates globally.

- **Debugging Distributed Flows**: Tracing a single request that spans multiple services and message queues can be incredibly difficult without proper tooling.
    - **Solution**: Implement **distributed tracing** (e.g., using OpenTelemetry, which is gaining widespread adoption as of 2026). Ensure robust logging with **correlation IDs** that are passed through every hop (HTTP headers, message attributes). Collect comprehensive metrics for your message broker and services.
- **Message Loss/Duplication**: While message queues are designed for reliability, misconfigurations or bugs can lead to lost or duplicated messages.
    - **Solution**: Ensure "at-least-once" delivery semantics from your message broker (most modern brokers provide this by default). Crucially, design consumer services to be **idempotent**, meaning processing a message multiple times has the same effect as processing it once. This is a fundamental principle for resilient asynchronous systems.
- **Over-engineering**: Introducing a message queue for a simple, low-volume interaction between two services often adds more operational overhead and complexity than it solves.
    - **Solution**: Start simple. Only introduce asynchronous patterns when the benefits (scalability, resilience, decoupling) clearly outweigh the added complexity. Don't add a message queue just because "microservices use queues."

Summary

In this chapter, we've navigated the crucial landscape of service-to-service communication, understanding the fundamental differences between synchronous and asynchronous approaches.

Here are the key takeaways:

- **Synchronous communication** is direct and blocking, offering immediate feedback but leading to tight coupling, increased latency, and potential cascading failures. It's best for immediate, short-lived, blocking operations where the caller must wait for a result.

- **Asynchronous communication** is non-blocking, often mediated by message queues or event buses, promoting loose coupling, high resilience, and better scalability. It's ideal for long-running tasks, high-throughput scenarios, and event-driven architectures, including complex AI agent workflows.
- **Tradeoffs are paramount**: Every architectural decision involves weighing the benefits (e.g., resilience, scalability) against the complexity and potential pitfalls (e.g., eventual consistency, debugging).
- **Hybrid approaches are common**: Real-world systems effectively combine both synchronous and asynchronous patterns to achieve optimal performance, resilience, and user experience.
- **Understand the "Why"**: Don't just pick a pattern; understand why it fits your specific problem, considering factors like immediate feedback needs, task duration, fault tolerance, and scalability goals.

As we move forward, we'll delve into specific patterns and technologies that enable these communication styles, such as message queues and event-driven systems, and explore how they contribute to building robust, scalable, and resilient distributed systems.

References

- **Microservices Architecture Style - Azure Architecture Center**: The official guide provides an excellent overview of microservices, including communication patterns and their implications.
    - https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices
- **Pattern: Asynchronous Request-Reply - Microsoft Learn**: Detailed explanation of a common asynchronous pattern for handling responses in non-blocking scenarios.
    - https://learn.microsoft.com/en-us/azure/architecture/patterns/async-request-reply
- **Apache Kafka Documentation**: Official documentation for a leading distributed streaming platform, widely used for high-throughput asynchronous event-driven communication.
    - https://kafka.apache.org/documentation/
- **RabbitMQ Documentation**: Official documentation for a popular open-source message broker, known for its robust message queuing capabilities.
    - https://www.rabbitmq.com/documentation.html
- **OpenTelemetry Project**: The official site for OpenTelemetry, a vendor-neutral standard for distributed tracing, metrics, and logging, crucial for observability in distributed systems.
    - https://opentelemetry.io/

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 04

Building Resilient Systems: Retries, Timeouts, and Circuit Breakers

Distributed systems are powerful, allowing us to scale applications and handle immense loads by breaking them into smaller, interconnected services. But here's a secret: they will fail. Networks are unreliable, services can crash, and dependencies can slow down. The real challenge isn't preventing all failures (an impossible task), but designing systems that can tolerate failures and continue to function gracefully.

This chapter dives into three fundamental patterns that form the bedrock of resilient distributed systems: **Retries**, **Timeouts**, and **Circuit Breakers**. You'll learn what each pattern is, why it's crucial, and how to apply it effectively to build applications that can withstand the chaos of a distributed environment. We'll also explore how these timeless principles are vital for emerging AI and agentic workflows, where interactions with external tools and models are frequent and often unpredictable.

By the end of this chapter, you'll have a robust mental model for designing systems that don't just work when everything is perfect, but truly shine when things go wrong.

The Inevitability of Failure in Distributed Systems

Imagine a single monolithic application. If it crashes, the whole thing stops. In a distributed system, you have many services talking to each other over a network. This introduces new complexities and failure points:

- **Network Latency and Dropped Packets:** Calls between services aren't instantaneous. Packets can get lost or delayed.
- **Service Unavailability:** A service might be temporarily down for maintenance, restarting, or overwhelmed by traffic.
- **Resource Exhaustion:** A service might run out of CPU, memory, or database connections, leading to unresponsiveness.
- **Partial Failures:** One part of your system might fail while others continue to operate, leading to inconsistent states or degraded functionality.

- **Dependency Failures:** A service might depend on another service that's currently failing, propagating the issue upstream.

These issues are not exceptions; they are the norm in complex systems. To build robust systems, we must embrace this reality and design for **fault tolerance** – the ability for a system to continue operating, perhaps in a degraded mode, even when some of its components fail.

Retry Pattern: Giving Operations a Second Chance

When you're trying to reach a friend on the phone and it goes straight to voicemail, what do you do? You probably try again a few minutes later, right? The **Retry pattern** applies this same common-sense approach to software.

What is the Retry Pattern?

The Retry pattern is a mechanism where a system re-attempts an operation that has previously failed. It's particularly useful for **transient failures** – those that are temporary and likely to resolve themselves quickly. Think of a brief network glitch, a database deadlock, or a service instance restarting.

Why Does it Exist?

Retries exist to increase the likelihood of success for operations affected by temporary issues without requiring manual intervention. They make your system more robust by smoothing over minor, short-lived disruptions, preventing unnecessary user-facing errors or workflow interruptions.

How Does it Work?

The simplest retry is just re-executing the failed call. However, a more sophisticated approach involves:

1. **Retry Count:** Defining how many times an operation should be re-attempted. Too many retries can be counterproductive, especially if the failure is persistent, potentially increasing load on an already struggling service.

2. **Delay/Backoff Strategy:** Waiting for a period before retrying. This prevents overwhelming the failing service and allows it time to recover.
    - **Fixed Delay:** Waiting the same amount of time between each retry (e.g., 1 second). Simple but less effective for heavily loaded services.
    - **Exponential Backoff:** Increasing the delay after each consecutive failure (e.g., 1s, then 2s, then 4s, then 8s). This is generally preferred as it gives the failing service exponentially more time to recover and reduces the rate of load.
    - **Jitter:** Adding a small, random amount of time to the delay. This is crucial with exponential backoff to prevent a "thundering herd" problem, where many clients retry at the exact same moment after an outage, causing a new surge of requests that overwhelms the recovering service. Jitter smooths out these retry attempts.

When to Use (and Not Use) Retries

Use Retries When:

- The operation is **idempotent**: executing it multiple times has the same effect as executing it once (e.g., updating a user's address to a specific value, rather than incrementing a counter). This is critical to avoid unintended side effects.
- The failure is **transient**: likely to resolve itself quickly.
- You are calling a **remote service or database** where transient network issues or temporary resource contention are common.

Avoid Retries When:

- The operation is **not idempotent**: retrying could lead to unintended side effects (e.g., processing the same payment twice, creating duplicate orders).
- The failure is **permanent**: retrying will never succeed (e.g., an invalid authentication token, a "resource not found" (404) error, or a validation error). In these cases, retries only waste resources.
- You risk creating a **thundering herd**: if many clients retry simultaneously without proper backoff/jitter, they can overwhelm a recovering service, preventing it from ever stabilizing.

⚡ **Real-world insight:** Retries are commonly implemented in HTTP client libraries, database drivers, and messaging queue consumers to handle transient network issues or temporary service unavailability. Many cloud SDKs (e.g., AWS Boto3, Azure SDK for Python) include built-in retry logic.

Timeout Pattern: Knowing When to Give Up

Imagine ordering food and waiting indefinitely. At some point, you'd give up and try another restaurant, right? The **Timeout pattern** is about setting an expectation for how long an operation should take.

What is the Timeout Pattern?

A timeout defines the maximum duration a client (or any component) is willing to wait for an operation to complete. If the operation doesn't finish within this time, it's aborted, and an error is returned. This prevents clients from getting stuck waiting forever.

Why Does it Exist?

Timeouts are essential for preventing a client from waiting indefinitely for a response, which can lead to:

- **Resource Exhaustion:** Holding open network connections, threads, or memory, consuming valuable system resources unnecessarily.
- **Poor User Experience:** Applications becoming unresponsive or extremely slow, frustrating users.
- **Cascading Failures:** A slow dependency holding up other services that rely on it, causing a ripple effect of unresponsiveness throughout the system.

How Does it Work?

Timeouts can be applied at various layers of your system:

- **Connection Timeout:** How long to wait to establish a connection (e.g., to a database or remote service). If the connection isn't made, it fails fast.
- **Read/Write Timeout (Socket Timeout):** How long to wait for data to be sent or received over an established connection. This catches cases where the connection is alive but the remote service isn't sending data.
- **Request Timeout (Total Timeout):** The total time allowed for an entire request-response cycle, from initiating the request to receiving the full response. This is often the most useful in application code.

Properly configured timeouts ensure that resources are released promptly and that your system can quickly detect and react to unresponsive dependencies.

When to Use Timeouts

Always use timeouts for:

- **Any network-bound operation:** HTTP requests, gRPC calls, database queries, message queue interactions.
- **Operations that involve external dependencies:** Third-party APIs, microservices, cloud services.
- **Long-running internal computations** that might get stuck or take an unexpectedly long time.

⚠️ **What can go wrong:**

- **Timeouts that are too short:** Leading to premature failures, even for healthy but slightly slow operations. This can reduce system availability.
- **Timeouts that are too long:** Still causing resource exhaustion and slow user experiences, defeating the purpose of the timeout.
- **Ignoring timeouts:** Allowing dependencies to hang indefinitely, causing your service to become unresponsive and potentially leading to cascading failures.

Choosing appropriate timeout values requires careful consideration of the expected latency of the dependency, the acceptable wait time for your application's users, and the service-level objectives (SLOs) you've defined.

Circuit Breaker Pattern: Preventing a Cascade

Retries help with transient failures, and timeouts prevent indefinite waits. But what if a dependency is permanently down or consistently failing? Continuously retrying or waiting for a timeout will just waste resources and further degrade system performance. This is where the **Circuit Breaker pattern** comes in.

What is the Circuit Breaker Pattern?

Inspired by electrical circuit breakers, this pattern prevents an application from repeatedly invoking a service that is likely to fail. When a service is deemed unhealthy, the circuit breaker "trips" (opens), immediately failing subsequent calls

to that service without attempting to execute them. This gives the failing service time to recover and prevents the calling service from wasting resources or experiencing cascading failures.

Why Does it Exist?

The Circuit Breaker pattern exists to:

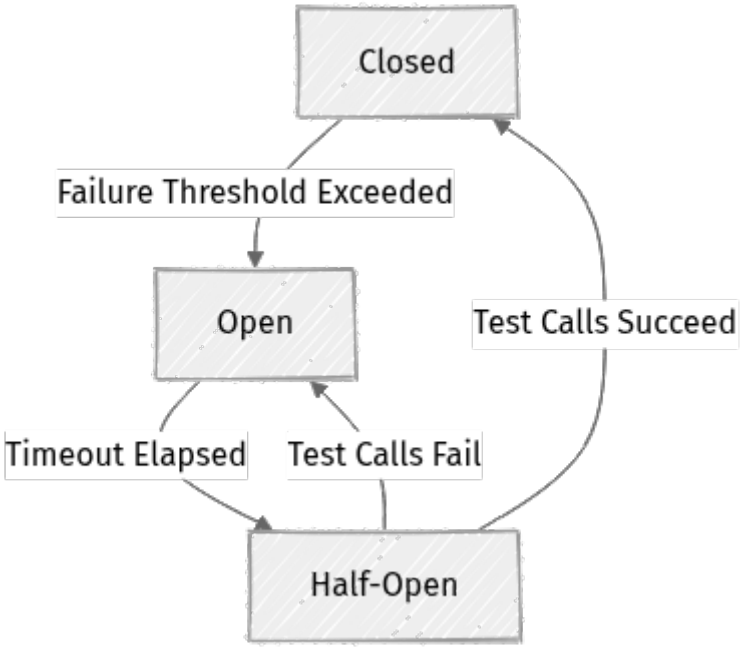
- **Protect the calling service:** By failing fast, it prevents the caller from consuming resources (threads, memory, network connections) waiting for a perpetually failing dependency.
- **Protect the called service:** By stopping requests, it gives the failing service a chance to recover without being overwhelmed by a flood of new requests from clients.
- **Prevent cascading failures:** A failing service can quickly bring down others that depend on it. The circuit breaker isolates the failure, containing it to a single component.

How Does it Work?

A circuit breaker typically operates in three states, acting as a state machine:

1. **Closed:** The default state. Calls to the service are allowed to pass through. The circuit breaker monitors for failures (e.g., exceptions, timeouts). If failures exceed a certain threshold within a defined period, the circuit moves to the **Open** state.
2. **Open:** Calls to the service are immediately rejected with an error (e.g., a `CircuitBreakerOpenException`). No actual calls are made to the unhealthy service. After a configurable "timeout" period (e.g., 30-60 seconds, which is the `reset_timeout`), the circuit automatically transitions to the **Half-Open** state.
3. **Half-Open:** A limited number of test requests are allowed to pass through to the protected service.
    - If these test requests succeed, the circuit assumes the service has recovered and moves back to the **Closed** state.
    - If they fail, the circuit returns to the **Open** state for another timeout period.

This state machine allows the system to intelligently adapt to the health of its dependencies.



⚡ **Real-world insight:** Circuit breakers are fundamental in microservice architectures, protecting services from slow or unresponsive dependencies. Libraries like Resilience4j in Java, Polly in .NET, `pybreaker` in Python, or various implementations in Go provide robust circuit breaker functionality. While Netflix Hystrix (a pioneering library) is deprecated, its core principles are widely adopted.

🧠 **Important:** A circuit breaker is not a retry mechanism. It stops retries to a failing service. It's about protecting the system as a whole, not just ensuring a single operation succeeds. When a circuit is open, you don't retry; you fail fast.

⚠️ **What can go wrong:**

- **Incorrect thresholds:** Too sensitive (trips too easily for minor glitches) or not sensitive enough (doesn't trip when it should, allowing failures to propagate).
- **Not resetting properly:** If the Half-Open state isn't configured correctly, the circuit might stay open or half-open longer than necessary, impacting service availability.
- **Ignoring the circuit breaker:** Simply retrying after a circuit breaker opens, defeating its purpose and potentially overwhelming the failing service.

Step-by-Step Implementation: Composing Resilience in Python

Let's see how these patterns can be combined in a practical Python example. We'll use the `requests` library for HTTP calls, `tenacity` for retries, and `pybreaker` for circuit breaking.

First, ensure you have these libraries installed:```bash pip install requests tenacity pybreaker

```
### 1. The Basic, Non-Resilient Call

Let's start with a simple function that calls an external API. For demonstration, we'll use `httpbin.org/status/500` to simulate a server error and `httpbin.org/delay/3` for a slow response.
```python
import requests

def call_external_service(url: str):
    """Makes a simple HTTP GET request without any resilience."""
    print(f"Attempting to call {url}...")
    try:
        response = requests.get(url)
        response.raise_for_status() # Raises HTTPError for bad responses (4xx
or 5xx)
        print(f"Success: {response.status_code}")
        return response.json()
    except requests.exceptions.RequestException as e:
        print(f"Failure: {e}")
        raise

# Example usage (will fail or hang)
# call_external_service("http://httpbin.org/status/500")
# call_external_service("http://httpbin.org/delay/3") # This will hang for 3
seconds
```

This function is fragile. A 500 error will immediately fail, and a slow response will block for the full duration.

2. Adding Timeouts

The first line of defense is a timeout. `requests` makes this easy with the `timeout` parameter. We'll set a total timeout for the request.

```
import requests

def call_external_service_with_timeout(url: str, timeout_seconds: float = 1.0)
:
    """Makes an HTTP GET request with a timeout."""
    print(f"Attempting to call {url} with timeout {timeout_seconds}s...")
    try:
        # The timeout parameter is for the entire request, including
connection and read
```
```

```
        response = requests.get(url, timeout=timeout_seconds)
        response.raise_for_status()
        print(f"Success: {response.status_code}")
        return response.json()
    except requests.exceptions.Timeout:
        print(f"Failure: Request timed out after {timeout_seconds}s!")
        raise
    except requests.exceptions.RequestException as e:
        print(f"Failure: {e}")
        raise

# Example usage:
# This will now fail quickly instead of hanging for 3 seconds
# call_external_service_with_timeout("http://httpbin.org/delay/3",
timeout_seconds=0.5)
# This will still fail on 500, but won't hang if the server is just slow to
respond
# call_external_service_with_timeout("http://httpbin.org/status/500")
```

Now, if `httpbin.org/delay/3` is called with a timeout of `0.5` seconds, it will fail fast with a `Timeout` exception instead of waiting the full 3 seconds.

3. Adding Retries with Exponential Backoff and Jitter

Next, we'll add retries for transient errors. We'll use the `tenacity` library, which provides decorators for this. We want to retry on `requests.exceptions.RequestException` (which includes timeouts and connection errors) and HTTP 5xx errors.

```
import requests
from tenacity import retry, wait_exponential, stop_after_attempt, retry_if_exc
eption_type, retry_if_result
import random

# Define what constitutes a retryable HTTP status code (e.g., 5xx errors)
def is_retryable_status_code(response):
    return response.status_code >= 500 if response is not None else False

@retry(
    wait=wait_exponential(multiplier=1, min=1, max=10),
# Exponential backoff: 1s, 2s, 4s, 8s...
    stop=stop_after_attempt(5), # Stop after 5 attempts
    # Retry on network errors, timeouts, or 5xx status codes
    retry=(
        retry_if_exception_type(requests.exceptions.RequestException) |
        retry_if_result(is_retryable_status_code)
    ),
    reraise=True # Re-raise the last exception if all retries fail
)
def call_external_service_with_retries(url: str, timeout_seconds: float = 1.0)
:
    """Makes an HTTP GET request with retries and a timeout."""
    print(f"Attempting to call {url} (retryable, timeout {timeout_seconds}s)..
.")
    try:
        response = requests.get(url, timeout=timeout_seconds)
```

```
        response.raise_for_status()
        print(f"Success: {response.status_code}")
        return response
    except requests.exceptions.Timeout:
        print(f"Failure: Request timed out after {timeout_seconds}s.
Retrying...")
        raise
    except requests.exceptions.RequestException as e:
        print(f"Failure: {e}. Retrying...")
        raise

# Example usage:
# This will retry 5 times with exponential backoff if httpbin.org/status/500
is called
# try:
#     call_external_service_with_retries("http://httpbin.org/status/500")
# except Exception as e:
#     print(f"Final failure after retries: {e}")

# Simulate a transient success (e.g., a service that's flaky)
# To test this, you'd need a service that fails sometimes and succeeds others.
# For example, if you ran a local server that returned 500 three times, then
200.
```

This significantly improves robustness against transient issues. `tenacity` automatically adds jitter by default with `wait_exponential`.

4. Adding a Circuit Breaker

Finally, we wrap our retryable, timeout-aware call with a circuit breaker using `pybreaker`. This protects against persistent failures.

```
import requests
from tenacity import retry, wait_exponential, stop_after_attempt, retry_if_exc
eption_type, retry_if_result
from pybreaker import CircuitBreaker, CircuitBreakerError
import random
import time

# --- Same retry logic as before ---
def is_retryable_status_code(response):
    return response.status_code >= 500 if response is not None else False

@retry(
    wait=wait_exponential(multiplier=1, min=1, max=10),
    stop=stop_after_attempt(5),
    retry=(
        retry_if_exception_type(requests.exceptions.RequestException) |
        retry_if_result(is_retryable_status_code)
    ),
    reraise=True
)
def _call_service_inner(url: str, timeout_seconds: float = 1.0):
    """Internal function for calling service with timeout and retries."""
    print(f"  --> Inner call attempt to {url} (timeout {timeout_seconds}
s)...")
    try:
        # Simulate a flaky service that sometimes fails permanently for a bit
```

```
        # This is for demonstration. In real life, the external service
decides.
        if "flaky" in url and random.random() < 0.8: # 80% chance of failure
            raise requests.exceptions.ConnectionError("Simulated connection
error for flaky service")

        response = requests.get(url, timeout=timeout_seconds)
        response.raise_for_status()
        print(f"  --> Inner call SUCCESS: {response.status_code}")
        return response
    except requests.exceptions.Timeout:
        print(f"  --> Inner call FAILURE: Request timed out. Retrying...")
        raise
    except requests.exceptions.RequestException as e:
        print(f"  --> Inner call FAILURE: {e}. Retrying...")
        raise

# --- Circuit Breaker setup ---
# Circuit breaker for the external service
# It will open if 3 consecutive calls fail (fail_max=3)
# It will stay open for 10 seconds (reset_timeout=10)
external_service_breaker = CircuitBreaker(fail_max=3, reset_timeout=10, exclud
e=[requests.exceptions.Timeout])

# Wrap the retryable function with the circuit breaker
@external_service_breaker
def call_external_service_resilient(url: str, timeout_seconds: float = 1.0):
    """Makes a resilient HTTP GET request with timeout, retries, and circuit
breaker."""
    print(f"Calling resilient service for {url}. Breaker state: {external_serv
ice_breaker.current_state}")
    try:
        return _call_service_inner(url, timeout_seconds)
    except Exception as e:
        print(f"  --> Resilient call encountered error: {e}")
        raise # Re-raise for the circuit breaker to count it as a failure

# --- Demonstrate the combined patterns ---
print("\n--- Scenario 1: Transient Failures (Retries handle it) ---")
# If httpbin.org/status/500 fails, tenacity will retry.
# If it eventually succeeds, the circuit breaker remains closed.
# For this example, let's assume _call_service_inner sometimes succeeds after
a few retries.
# In a real scenario, httpbin.org/status/500 would just keep failing.
# To truly demonstrate, we'd need a mock server that sometimes returns 500,
then 200.
# Let's use a URL that we expect to succeed eventually.
try:
    # A URL that is generally reliable
    call_external_service_resilient("http://httpbin.org/get",
timeout_seconds=0.5)
except CircuitBreakerError:
    print("Circuit breaker is open! Not trying again.")
except Exception as e:
    print(f"Final failure: {e}")

print("\n--- Scenario 2: Persistent Failures (Circuit Breaker trips) ---")
# This will likely cause the circuit breaker to trip because
_call_service_inner
# will often simulate failure for "flaky" URLs.
for i in range(10):
    try:
```

```
        call_external_service_resilient("http://flaky-service.example.com/
api", timeout_seconds=0.5)
        time.sleep(0.1) # Small delay between calls
    except CircuitBreakerError:
        print(f"Attempt {i+1}: Circuit breaker is OPEN. Not making call.")
        time.sleep(1) # Wait a bit before next attempt to see half-open state
    except Exception as e:
        print(f"Attempt {i+1}: Call failed with {type(e).__name__}. Breaker
state: {external_service_breaker.current_state}")
        time.sleep(0.1) # Small delay

print("\n--- Scenario 3: Circuit Breaker Half-Open State ---")
print(f"Waiting for reset_timeout ({external_service_breaker.reset_timeout}s)
for breaker to go Half-Open...")
time.sleep(external_service_breaker.reset_timeout + 1) # Wait for the
reset_timeout to pass

try:
    # This call will be a test call in the Half-Open state
    call_external_service_resilient("http://httpbin.org/get",
timeout_seconds=0.5)
    print("Test call successful. Circuit should now be CLOSED.")
except CircuitBreakerError:
    print("Test call failed. Circuit remains OPEN.")
except Exception as e:
    print(f"Test call failed with {type(e).__name__}. Circuit remains OPEN.")

print(f"Final Breaker State: {external_service_breaker.current_state}")
```

In this setup:

1. The `_call_service_inner` function includes **timeouts** and **retries** for transient failures.
2. The `call_external_service_resilient` function wraps `_call_service_inner` with a **circuit breaker**.
3. If `_call_service_inner` consistently fails (even after its own retries), the circuit breaker will trip, preventing further attempts for a period.
4. Notice how we use `exclude=[requests.exceptions.Timeout]` on `CircuitBreaker`. This is a subtle but important detail: if a timeout always means the service is slow but not down, you might not want it to trip the circuit breaker immediately. Often, though, you do want timeouts to contribute to circuit breaker failure counts, so you'd remove `exclude`. For this example, we're showing flexibility.

This layered approach creates a highly resilient interaction between services, gracefully handling various failure types.

Resilience in AI/Agent Workflows

The principles of retries, timeouts, and circuit breakers are even more critical in modern AI and agentic systems. These systems frequently interact with external components, often with unpredictable latency and reliability:

- **Large Language Models (LLMs):** API calls to models (e.g., OpenAI, Anthropic, Google Gemini) can experience rate limits, temporary unavailability, or high latency due to heavy load or network issues.
- **External Tools/APIs:** Agents often use tools that wrap external web services (e.g., weather APIs, payment gateways, search engines, specialized data services). These are prone to all the distributed system issues we've discussed.
- **Vector Databases/Knowledge Bases:** Interactions with these data stores for retrieval-augmented generation (RAG) or memory management need to be robust.
- **Orchestration Services:** Coordinating multiple agents or steps in a complex workflow requires resilient communication between internal components.

Imagine an AI agent designed to book travel. It might:

1. Call a flight search API.
2. Call a hotel booking API.
3. Call a payment gateway.

Each of these steps is an external dependency. If the flight search API has a brief outage, retries can ensure the agent eventually gets results. If the hotel booking API is consistently slow, a timeout prevents the agent from hanging indefinitely, allowing it to inform the user or try an alternative. If the payment gateway is completely down, a circuit breaker prevents the agent from repeatedly failing payment attempts, potentially causing issues or wasting resources.

By integrating these resilience patterns, AI agents can become more robust, reliable, and capable of operating effectively even when their underlying tools and models encounter transient or persistent issues.

Mini-Challenge: Design a Resilient AI Tool Call

You are building an AI agent that needs to interact with a third-party image generation API. This API is known to occasionally have transient network issues and sometimes experiences longer outages during peak times. The API is not idempotent for image generation requests (retrying a successful request might generate a duplicate image with a slightly different ID).

Challenge: Describe, in plain language, how you would integrate retries, timeouts, and a circuit breaker into your agent's code to make calls to this image generation API resilient. Focus on the logic flow and why you're applying each pattern. Specifically address the non-idempotent nature of the API.

Hint: Think about the order in which these patterns would "wrap" the API call. What are reasonable values or strategies for each? How does non-idempotency affect your retry strategy?

What to Observe/Learn: This exercise helps you solidify your understanding of how these patterns combine to form a comprehensive resilience strategy for real-world scenarios, and how to adapt them to specific API characteristics like idempotency.

Common Pitfalls & Troubleshooting

Even with these powerful patterns, misconfigurations can lead to new problems:

- **Over-aggressive Retries (The Thundering Herd):** If many clients retry simultaneously without sufficient backoff and jitter, they can overwhelm a recovering service, preventing it from ever fully recovering. This is a common cause of service instability after an outage.
- **Incorrect Timeout Values:**
    - **Too short:** Leading to unnecessary failures for operations that would have succeeded given a little more time. This can make your system appear less available than it is.
    - **Too long:** Still causing resource exhaustion and poor user experience during slow responses. This can lead to cascading slowness.
- **Ignoring Circuit Breaker State:** If a client doesn't respect an open circuit breaker and tries to bypass it (e.g., by manually retrying regardless), the purpose of the pattern is defeated, and the failing service remains under stress.

- **Lack of Observability:** Without proper logging, metrics, and tracing (which we'll cover in a later chapter!), it's incredibly difficult to know why a circuit breaker tripped, why retries failed, or what the actual latency of a dependency is. This makes tuning resilience parameters a guessing game and debugging a nightmare.
- **Resilience Configuration Drift:** Different services calling the same dependency might have different resilience settings, leading to inconsistent behavior and making it hard to predict how the system will react under stress.
- **Retrying Non-Idempotent Operations:** As highlighted in the mini-challenge, blindly retrying operations that are not idempotent can lead to duplicate data, double charges, or other undesirable side effects. Always understand the idempotency characteristics of the operations you are retrying.

Troubleshooting resilience issues often involves looking at logs from both the calling and called services, monitoring network latency, analyzing performance metrics over time, and carefully reviewing the configuration of your resilience patterns.

Summary

Building resilient systems is not about avoiding failures, but about intelligently handling them. In this chapter, we've explored three cornerstone patterns:

- **Retries:** Give operations a second (or third, or fourth) chance for transient failures, using strategies like exponential backoff and jitter to prevent overwhelming services.
- **Timeouts:** Prevent indefinite waits and resource exhaustion by setting clear limits on operation duration, ensuring resources are released promptly.
- **Circuit Breakers:** Isolate failing dependencies, protecting both the caller and the callee from cascading failures and allowing services time to recover. They implement a state machine to intelligently manage interaction with unhealthy services.

These patterns, when combined thoughtfully, allow your applications to be more robust, reliable, and capable of gracefully navigating the inherent unpredictability of distributed environments, including the complex and often flaky interactions within modern AI and agentic workflows. Understanding their purpose, implementation, and common pitfalls is crucial for any systems engineer.

In the next chapter, we'll shift our focus to **queues and asynchronous workflows**, exploring how they further enhance scalability and resilience by decoupling services and managing tasks that don't require immediate responses.

References

- Microservices Architecture Style - Azure Architecture Center
- Circuit Breaker Pattern - Microsoft Azure Architecture Center
- Retry Pattern - Microsoft Azure Architecture Center
- Timeouts, Retries, and Circuit Breakers with HTTP - Martin Fowler
- Tenacity (Python retry library) Documentation
- Pybreaker (Python circuit breaker library) Documentation

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant. +++

CHAPTER 05

Decoupling Services with Message Queues and Asynchronous Workflows

Introduction: Breaking Free from Tight Coupling

Imagine a bustling restaurant where every customer order is taken by a chef directly, cooked immediately, and then the chef waits for the customer to finish before taking the next order. This is what synchronous, tightly coupled services often feel like in a software system. If one chef is busy or sick, the whole kitchen grinds to a halt. Not very efficient or resilient, right?

In the world of distributed systems, services frequently need to communicate. While direct, synchronous API calls (like HTTP requests) are common, they can introduce significant dependencies. If Service A calls Service B, and Service B is slow or unavailable, Service A often has to wait or fail. This tight coupling limits scalability, reduces fault tolerance, and makes independent deployment a nightmare.

This chapter dives into the transformative power of **message queues** and **asynchronous workflows**. You'll learn how these timeless engineering principles allow services to communicate without waiting for immediate responses, building systems that are more resilient, scalable, and easier to evolve. We'll explore the core concepts, practical applications, and how even advanced AI agent systems leverage these patterns for distributed task execution.

The Core Problem: Synchronous Bottlenecks

Before we embrace asynchronous patterns, let's solidify our understanding of the problem they solve.

When services communicate synchronously, they essentially block and wait.

- **Service A** sends a request to **Service B**.
- **Service A** pauses its current execution, waiting for **Service B** to respond.
- **Service B** processes the request and sends a response.
- **Service A** resumes its execution with the response.

This direct dependency works for simple cases, but it quickly becomes a bottleneck as systems grow.

⚠️ **What can go wrong:**

- **Cascading Failures:** If Service B fails, Service A (and any service calling A) might also fail.
- **Reduced Throughput:** Service A can only process as fast as Service B can respond.
- **Latency Spikes:** Network delays or slow processing in Service B directly impact Service A's response time.
- **Deployment Complexity:** Service A and B often need to be deployed and scaled together, or at least in a specific order, making updates harder.

Message Queues: The Ultimate Decoupler

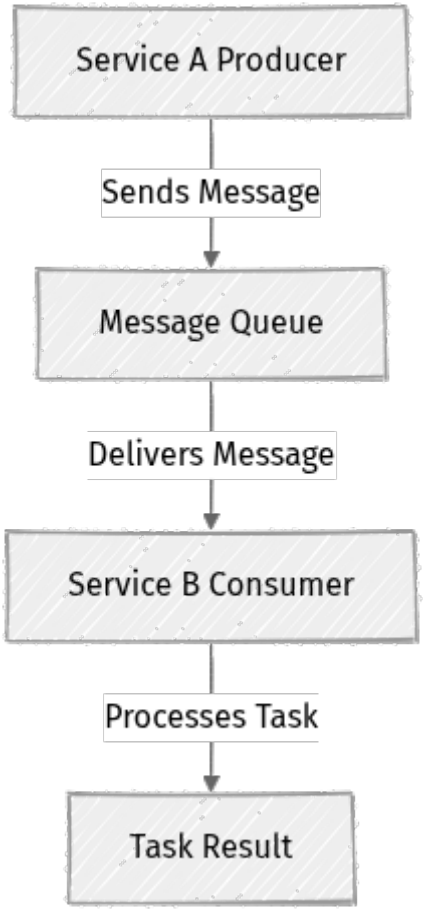
Enter the **message queue**. Think of it as a highly reliable post office for your services. Instead of directly calling another service and waiting, a service simply drops a "letter" (a message) into a queue. Another service, at its own pace, picks up letters from the queue and processes them.

What is a Message Queue?

A message queue is a form of asynchronous service-to-service communication used in serverless and microservices architectures. It's a temporary repository where messages wait to be processed.

The core components are:

- **Producer (Sender):** A service that creates and sends messages to the queue.
- **Consumer (Receiver/Worker):** A service that retrieves messages from the queue and processes them.
- **Queue (Broker):** The intermediary component that stores messages until consumers are ready to process them. It handles message delivery, durability, and often ordering.



Why Use Message Queues?

Message queues solve the synchronous coupling problem by introducing an intermediary.

1. **Decoupling:** Producers don't need to know anything about consumers, and vice-versa, beyond the message format. They operate independently.
2. **Asynchronous Processing:** Producers send messages and immediately continue their work, without waiting for the message to be processed. This is crucial for long-running tasks.
3. **Buffering and Load Leveling:** If consumers are busy, messages accumulate in the queue. When consumers become free, they pull messages from the queue. This prevents producers from overwhelming consumers and smooths out traffic spikes.
4. **Resilience:** If a consumer fails, messages remain in the queue, ready for another consumer (or the restarted consumer) to pick them up. This improves fault tolerance.
5. **Scalability:** You can easily add more consumers to process messages faster, scaling horizontally based on demand without impacting producers.

6. **Reliability:** Most message queues ensure "at-least-once" delivery, meaning a message won't be lost even if a consumer fails mid-processing.

How Message Queues Work: A Simple Flow

Let's trace a typical message flow:

1. **Produce:** A service generates a message (e.g., "New user registered with ID 123"). It then sends this message to a designated queue.
2. **Store:** The message queue broker receives the message and stores it durably.
3. **Consume:** A consumer service polls the queue or receives messages pushed from the queue. It retrieves the message.
4. **Process:** The consumer processes the message (e.g., sends a welcome email to user 123).
5. **Acknowledge:** After successful processing, the consumer sends an acknowledgment back to the queue.
6. **Delete:** Upon acknowledgment, the queue broker deletes the message, knowing it's been handled.

🧠 **Important:** If a consumer fails before acknowledging, the message typically becomes visible again after a timeout, allowing another consumer (or the same consumer once recovered) to retry processing. This mechanism ensures messages aren't lost.

When to Use Message Queues

Message queues are not a silver bullet for all communication, but they excel in specific scenarios:

- **Long-Running Tasks:** Any operation that takes more than a few hundred milliseconds (e.g., image processing, video transcoding, report generation, complex AI model inference).
- **Fan-out Operations:** When one event needs to trigger multiple independent actions (e.g., a new user registration triggers sending a welcome email, updating a CRM, and provisioning user resources).
- **Batch Processing:** Processing large volumes of data in chunks.
- **Cross-Service Communication in Microservices:** Decoupling services that don't require immediate, synchronous responses.
- **Event-Driven Architectures:** Queues are a foundational component for distributing events, which we'll cover in the next chapter.

- **AI Agent Task Distribution:** Distributing tasks to multiple AI agents, collecting their results, and managing long-running agentic workflows. For example, an orchestrator agent might place tasks for specialized agents (e.g., "research topic X," "summarize article Y") into a queue, and worker agents pick them up.

When NOT to Use Message Queues:

- **Real-time Request-Response:** For operations where the client must receive an immediate response from the server (e.g., retrieving user profile data, login authentication). Introducing a queue here would add unnecessary latency and complexity.
- **Simple, Low-Volume Synchronous Calls:** If services are already tightly coupled and highly reliable, and the overhead of a queue outweighs the benefits.
- **When Strict Ordering is Paramount and Complex:** While some queues offer strict ordering guarantees, implementing end-to-end exactly-once processing with complex ordering requirements can be challenging and might introduce bottlenecks.

Worker Architectures and Asynchronous Workflows

Message queues enable powerful **worker architectures**. A worker service is essentially a consumer that continuously pulls tasks (messages) from a queue and performs specific work.

For example, an e-commerce order processing system might have:

- **Order Service (Producer):** Receives a new order, saves it to the database, and sends an "Order Placed" message to a queue.
- **Payment Processor Worker (Consumer):** Picks up "Order Placed" messages, processes the payment, and sends a "Payment Processed" message to another queue.
- **Shipping Worker (Consumer):** Picks up "Payment Processed" messages, initiates shipping, and sends a "Shipment Initiated" message.

This chain of events forms an **asynchronous workflow**. These workflows can be complex, involving multiple queues and worker types.

⚡ **Real-world insight:** Many cloud providers offer managed message queue services (e.g., AWS SQS/SNS, Azure Service Bus, Google Cloud Pub/Sub). These services handle the operational complexities of running a message broker, allowing you to focus on your application logic. As of 2026, these services are mature and highly reliable.

AI/Agentic Systems and Message Queues

AI and agentic systems frequently leverage message queues to manage their distributed nature:

- **Task Distribution:** A central "orchestrator" agent or service can place complex tasks into a queue. Multiple "worker" agents (specialized for different functions like data retrieval, code generation, summarization) can then pick up and process these tasks in parallel.
- **Long-Running Agentic Workflows:** If an agent's reasoning or action might take significant time, the intermediate steps can be placed in a queue. For instance, an agent deciding to "research a topic" could put a `research_task` message into a queue, and a dedicated research agent picks it up. Once done, the research agent puts a `research_result` message into another queue for the original agent to continue its workflow.
- **Result Aggregation:** When multiple agents work on sub-tasks, their results can be sent to a queue, from which an aggregation service or another agent can collect and synthesize the final outcome.
- **Rate Limiting and Backpressure:** Queues naturally help manage the load on computationally intensive AI models or external APIs by buffering requests.

Step-by-Step Illustration: A Simple Asynchronous Task

Let's illustrate the producer-consumer pattern using conceptual pseudocode for a hypothetical `ImageProcessingQueue`.

Step 1: Define the Message Structure

First, consider what information your message needs to carry. For image processing, it might be the image ID and the type of processing requested.

```
# Conceptual Message Structure
{
  "image_id": "uuid-of-the-image-123",
```

```
  "operation": "resize",
  "parameters": {
    "width": 800,
    "height": 600
  },
  "callback_url": "https://myapp.com/api/image-processed-webhook"
}
```

Explanation: This JSON-like structure defines the payload for our image processing task. `image_id` identifies the target, `operation` specifies the task, `parameters` provides details for that task, and `callback_url` is an optional field for notifying the original service when processing is complete.

Step 2: The Producer Service (Sends Tasks)

This service is responsible for initiating the image processing. It doesn't perform the processing itself; it just tells the queue what needs to be done.

```python
# Conceptual Python-like Pseudocode for a Producer
import message_queue_client # Assume this is a library for your queue service

def upload_image(image_data):
    image_id = save_image_to_storage(image_data) # Stores image, returns ID
    print(f"Image uploaded with ID: {image_id}")

    # Create the message
    task_message = {
        "image_id": image_id,
        "operation": "resize",
        "parameters": {"width": 800, "height": 600},
        "callback_url": "https://myapp.com/api/image-processed-webhook"
    }

    # Send the message to the queue
    queue_name = "image_processing_tasks"
    message_queue_client.send_message(queue_name, task_message)
    print(f"Sent resize task for image {image_id} to queue '{queue_name}'")

    return {"status": "processing_initiated", "image_id": image_id}

# Example usage:
# response = upload_image(some_image_bytes)
# print(response)
```

Explanation:

1. `save_image_to_storage(image_data)`: This function (hypothetical) would handle storing the raw image data, perhaps in an object storage like AWS S3 or Azure Blob Storage, and return a unique ID.
2. `task_message`: We construct the message payload based on our defined structure.

3. `message_queue_client.send_message(...)`: This is the crucial part. The producer uses a client library to connect to the message queue broker and send the `task_message` to the specified `image_processing_tasks` queue.
4. The producer immediately returns a response, indicating that processing has started, not necessarily finished. This is the essence of asynchronous communication.

Step 3: The Consumer Service (Processes Tasks)

This service runs continuously, listening for new messages on the `image_processing_tasks` queue.

```
# Conceptual Python-like Pseudocode for a Consumer (Worker)
import message_queue_client # Assume this is a library for your queue service
import image_processor_library # Library for actual image manipulation
import requests # For sending callback

def process_image_task(message):
    image_id = message["image_id"]
    operation = message["operation"]
    parameters = message["parameters"]
    callback_url = message.get("callback_url") # Get with default None if not
present

    print(f"Received task: {operation} image {image_id} with params {parameter
s}")

    try:
        # 1. Download image from storage using image_id
        image_data = download_image_from_storage(image_id)

        # 2. Perform the actual image processing
        if operation == "resize":
            processed_image_data = image_processor_library.resize(image_data,
parameters["width"], parameters["height"])
        elif operation == "grayscale":
            processed_image_data = image_processor_library.to_grayscale(image_
data)
        else:
            raise ValueError(f"Unknown operation: {operation}")

        # 3. Save the processed image
        processed_image_url = save_processed_image(image_id, processed_image_d
ata, operation)
        print(f"Successfully processed image {image_id}. Result URL: {processe
d_image_url}")

        # 4. Notify original service if callback URL is provided
        if callback_url:
            requests.post(callback_url, json={
                "image_id": image_id,
                "status": "completed",
                "result_url": processed_image_url
            })
            print(f"Sent completion callback for image {image_id}")
```

```
        return True # Task processed successfully
    except Exception as e:
        print(f"Error processing image {image_id}: {e}")
        # In a real system, you'd log this error and potentially move the
message to a DLQ
        return False # Task failed

def start_worker():
    queue_name = "image_processing_tasks"
    print(f"Worker started, listening on queue '{queue_name}'...")
    while True:
        message = message_queue_client.receive_message(queue_name)
        if message:
            success = process_image_task(message)
            if success:
                message_queue_client.acknowledge_message(queue_name, message)
            else:
                # Depending on queue, message might be returned after timeout,
                # or manually moved to a Dead Letter Queue (DLQ)
                print(f"Task for message {message['image_id']} failed, not
acknowledging immediately.")
        else:
            time.sleep(5) # Wait if no messages

# To run:
# start_worker()
```

Explanation:

1. `start_worker()`: This function sets up an infinite loop to continuously check the queue for messages.
2. `message_queue_client.receive_message(...)`: The consumer pulls a message from the queue. This call might block until a message is available or return `None` after a timeout.
3. `process_image_task(message)`: This function contains the core business logic. It downloads the image, performs the requested operation, saves the result, and optionally sends a notification back to the original service via a webhook.
4. `message_queue_client.acknowledge_message(...)`: **Crucially**, after the task is successfully completed, the consumer acknowledges the message. This tells the queue broker that the message has been handled and can be safely removed.
5. Error Handling: If an error occurs during `process_image_task`, the message is not acknowledged immediately. This allows the message to become visible again later for a retry, or eventually move to a Dead Letter Queue (DLQ) if retries fail.

Mini-Challenge: Designing an AI Agent Workflow

You're building an AI platform where a "Master Agent" can delegate complex analytical tasks to specialized "Analysis Agents." Design a conceptual asynchronous workflow for the following scenario:

Scenario: A user uploads a document, and the Master Agent needs to perform three independent analyses on it: sentiment analysis, entity extraction, and summarization. Each analysis is done by a different, specialized Analysis Agent. The Master Agent then needs to collect all three results to synthesize a final report.

Your Task:

1. **Identify the Producers:** Which service(s) will send messages?
2. **Identify the Consumers (Worker Agents):** Which services/agents will process messages?
3. **Identify the Queues:** How many queues would you use, and for what purpose?
4. **Outline the Message Flow:** Describe the sequence of messages and processing steps from document upload to final report generation.

Hint: Think about how the Master Agent gets notified when all sub-tasks are complete. You might need more than one queue.

Common Pitfalls & Troubleshooting

Working with asynchronous systems and message queues introduces new complexities. Being aware of these common pitfalls can save you a lot of headaches.

1. **Ignoring Dead Letter Queues (DLQs):**
    - **What it is:** A DLQ is a special queue where messages are sent if they fail to be processed successfully after a certain number of retries, or if they expire.
    - **What can go wrong:** Without a DLQ, failed messages are simply discarded, leading to data loss or unaddressed errors.
    - **Pro tip:** Always configure a DLQ for critical queues. Monitor your DLQs and have a process to inspect, fix, and re-process messages from them. This is vital for data integrity.

2. **Lack of Idempotency in Consumers:**
    - **What it is:** An operation is idempotent if executing it multiple times produces the same result as executing it once.
    - **Why it matters:** Message queues often guarantee "at-least-once" delivery. This means a consumer might receive and process the same message multiple times (e.g., if it processes but fails to acknowledge before a timeout).
    - **What can go wrong:** If your consumer isn't idempotent, processing the same message twice could lead to duplicate charges, incorrect data updates, or other undesirable side effects.
    - **Solution:** Design your consumer logic to handle duplicate messages gracefully. This often involves checking for uniqueness (e.g., using a message ID or a unique business ID within the message) before performing state-changing operations.
3. **Message Ordering Guarantees:**
    - **What it is:** The order in which messages are delivered to consumers.
    - **What can go wrong:** Most general-purpose message queues do not guarantee strict message order across multiple consumers. If message A is sent before message B, it's possible consumer 1 processes B before consumer 2 processes A. If your business logic requires strict ordering (e.g., "deposit $10" must happen before "withdraw $5"), relying on default queue behavior can lead to incorrect states.
    - **Solution:** For strict ordering, consider using queues with specific ordering features (e.g., Kafka topics with partitions, AWS SQS FIFO queues) or design your application to handle out-of-order messages by making operations idempotent and using versioning or timestamps within your messages.

4. **Undersestimating Operational Overhead:**
    - **What it is:** While managed queue services simplify things, you still need to monitor queue depth, message latency, consumer error rates, and DLQ activity.
    - **What can go wrong:** Unmonitored queues can silently accumulate messages, leading to processing delays, resource exhaustion, or data loss if not addressed.
    - **Pro tip:** Integrate queue metrics into your observability stack. Set up alerts for high queue depth, high message age, or frequent consumer errors.

Summary: Building Robust Asynchronous Foundations

Congratulations! You've taken a significant step in understanding how to build more robust and scalable distributed systems. We covered:

- The limitations of **synchronous communication** and the need for **decoupling**.
- The core components and benefits of **message queues** for asynchronous communication, buffering, and resilience.
- How **worker architectures** consume messages to perform tasks.
- The role of queues in enabling complex **asynchronous workflows**, especially in **AI agent systems** for task distribution and result aggregation.
- Key considerations like **idempotency**, **message ordering**, and the importance of **Dead Letter Queues** for handling failures.

By thoughtfully applying message queues, you can design systems that gracefully handle variable loads, recover from failures, and allow services to evolve independently. This is a fundamental pattern for any large-scale, resilient architecture.

In the next chapter, we'll build upon this foundation to explore **event-driven systems**, where messages become "events" that drive reactive behaviors across your entire architecture.

References

- Azure Architecture Center - Microservices architecture style
- AWS SQS Developer Guide
- Google Cloud Pub/Sub Documentation
- Apache Kafka Documentation

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 06

Worker Architectures: Designing for Background Processing and Scalability

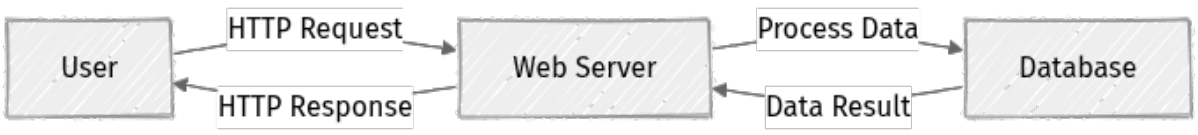
Imagine your application needs to perform a task that takes a long time – perhaps generating a complex report, processing a large image, or training a small AI model. If your user has to wait for this task to complete before they can do anything else, they'll likely get frustrated and leave. This is where worker architectures come into play, transforming slow, blocking operations into smooth, scalable background processes.

In this chapter, we'll dive into the world of worker architectures, understanding how they decouple long-running tasks from your main application flow. We'll explore the core components that make these systems robust and scalable, and discuss how timeless engineering principles like idempotency and error handling are critical for their success. By the end, you'll be able to design systems that handle heavy loads gracefully, ensuring a responsive user experience and efficient resource utilization, especially relevant for today's AI-driven applications.

If you've been following along, you'll find that the concepts of service-to-service communication and message queues from previous chapters provide an excellent foundation for understanding how workers communicate and manage tasks. We're building on those ideas to create even more powerful and resilient systems.

The Problem with Synchronous Processing

Most web applications start with a synchronous request-response model. A user makes a request, the server processes it, and then sends a response back. This works perfectly for fast operations like fetching data or simple form submissions.



However, what happens when "Process Data" takes 10 seconds, 30 seconds, or even several minutes? ⚠️ What can go wrong:

- **User Experience:** The user interface freezes or displays a loading spinner for an unacceptably long time, leading to frustration.

- **Timeouts:** Web servers and proxies often have timeout limits. If the task exceeds these, the connection drops, and the user receives an error, even if the task might eventually succeed.
- **Resource Blocking:** While one long task runs, server resources are tied up, potentially preventing other users from being served efficiently.
- **Scalability Challenges:** It's hard to scale a system where critical resources are blocked by slow operations.

This is a fundamental limitation for applications with complex backend logic, especially those incorporating AI/ML tasks that can be computationally intensive.

Introducing Worker Architectures

Worker architectures solve the synchronous processing problem by introducing **asynchronous task execution**. Instead of immediately processing a long-running task, your main application offloads it to a separate component: a worker.

📌 Key Idea: Decouple task initiation from task execution.

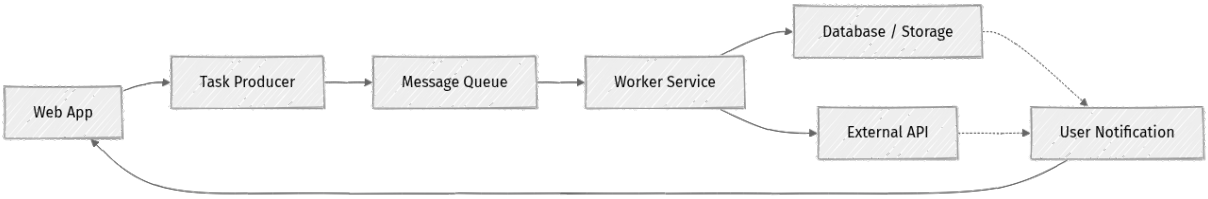
Think of it like this: You go to a busy coffee shop. You place your order (a task), and the barista (your main application) doesn't immediately start grinding beans and making your latte. Instead, they write your order on a ticket and put it into a queue. A dedicated coffee maker (the worker) then picks up tickets from the queue, makes the coffee, and calls your name when it's ready. You're free to wait, browse your phone, or do other things while your coffee is being prepared.

Why Worker Architectures Exist

- **Responsiveness:** Your main application (e.g., web server) can immediately respond to the user, confirming that their request has been received and will be processed. This vastly improves user experience.
- **Scalability:** Workers can be scaled independently of your main application. If you have a sudden surge of long-running tasks, you can spin up more workers to handle the load without affecting your web servers.
- **Resilience:** If a worker fails while processing a task, the task can often be retried by another worker, or moved to a dead-letter queue for inspection. This makes the system more fault-tolerant.
- **Efficiency:** Workers can be optimized for specific types of tasks, using different resource configurations (e.g., more memory, GPU access) than your frontend servers.

Key Components of a Worker System

A typical worker architecture involves three main players: a **Task Producer**, a **Message Queue**, and **Worker Services**.



⚡ Quick Note: The user notification step (G) often happens via another message back to the Web App or directly to the user, indicating task completion.

1. Task Producer

The task producer is the part of your application that identifies a long-running operation and decides to offload it. This is typically your frontend web server, an API endpoint, or another microservice.

When a producer encounters a task that shouldn't block the main thread, it creates a "message" describing the task and sends it to a message queue.

Example:

- A user uploads a high-resolution image. The web server (producer) doesn't resize it immediately.
- An AI agent needs to analyze a large text document. The orchestrating service (producer) doesn't perform the analysis itself.

2. Message Queue

The message queue is the central nervous system of a worker architecture. It acts as a buffer and a communication channel between producers and workers.

What it is: A durable, ordered list of messages (tasks) waiting to be processed.
Why it exists:

- **Decoupling:** Producers and consumers don't need to know about each other's existence or availability. They only interact with the queue.
- **Buffering:** It absorbs spikes in traffic. If producers generate tasks faster than workers can process them, the queue holds the tasks until workers become available.
- **Durability:** Messages can be persisted on disk, so they aren't lost if the queue service or a worker crashes.

- **Load Leveling:** Distributes tasks evenly among available workers.

Popular message queue technologies as of 2026-05-15 include:

- **Apache Kafka:** High-throughput, distributed streaming platform, often used for real-time data pipelines and event streaming. The latest stable release is 3.7.0.
- **RabbitMQ:** A robust, general-purpose message broker supporting various messaging protocols. The latest stable release is 3.13.0.
- **AWS SQS (Simple Queue Service):** A fully managed message queuing service by Amazon Web Services, known for its scalability and ease of use.
- **Azure Service Bus:** A fully managed enterprise integration message broker by Microsoft Azure, offering reliable message delivery.

3. Worker Service(s)

The worker service is the dedicated application or set of applications responsible for consuming messages from the queue and performing the actual work. Workers are often stateless, meaning they don't hold conversational state between tasks, making them easy to scale horizontally.

How it works:

1. A worker connects to the message queue.
2. It continuously polls or subscribes to the queue for new messages.
3. When a message arrives, the worker "claims" it (often making it invisible to other workers).
4. The worker processes the task described in the message.
5. Upon successful completion, the worker "acknowledges" the message, removing it from the queue.
6. If the worker fails, the message might be returned to the queue after a timeout, or moved to a Dead-Letter Queue (DLQ).

`⚡ Real-world insight:` In production, you'll often have multiple instances of your worker service running simultaneously, each pulling tasks from the same queue. This provides both high availability and scalability.

Designing Robust Workers

Building a basic worker is straightforward, but building a robust worker requires careful consideration of potential failure modes and edge cases in a distributed system.

Idempotency: The Superpower of Retries

What it is: An operation is idempotent if performing it multiple times has the same effect as performing it once. **Why it's important:** In distributed systems, messages can be delivered multiple times (e.g., due to network issues, worker restarts, or queue retries). If your task is not idempotent, processing a message twice could lead to incorrect data or unintended side effects.

Example:

- **Non-idempotent:** `increment_user_balance(user_id, amount)` - If run twice, the user's balance is incorrectly incremented twice.
- **Idempotent:** `set_user_balance_to(user_id, new_balance)` - Running this multiple times with the same `new_balance` has the same final effect.
- **Idempotent (with unique ID):** `process_order(order_id)` - If the worker first checks if `order_id` has already been processed and only proceeds if not, it becomes idempotent. This often involves storing a record of processed task IDs.

🧠 Important: Always design your worker tasks to be idempotent where possible. If not, implement mechanisms to detect and prevent duplicate processing using unique transaction IDs or correlation IDs.

Error Handling & Retries

Workers will inevitably encounter errors. How you handle them is crucial for system reliability.

- **Transient Errors:** Temporary issues like network glitches, database connection drops, or a third-party API being momentarily unavailable. For these, **retries** are essential.
    - **Exponential Backoff:** Instead of retrying immediately, wait for increasing intervals (e.g., 1s, 2s, 4s, 8s) before retrying. This prevents overwhelming the failing resource and gives it time to recover.
    - **Max Retries:** Set a limit on the number of retries to prevent infinite loops for persistent errors.

- **Persistent Errors:** Errors that won't resolve on their own, like invalid input data or a bug in the worker code.
    - **Dead-Letter Queues (DLQs):** If a message fails after multiple retries, it should be moved to a DLQ. This keeps "poison messages" from blocking the main queue and provides a place for manual inspection and debugging.
    - **Alerting:** Set up alerts when messages land in a DLQ so operations teams can investigate.

Concurrency and Resource Management

Workers need to manage how many tasks they process concurrently.

- **Too few workers/low concurrency:** Tasks build up in the queue, leading to delays.
- **Too many workers/high concurrency:** Workers consume excessive CPU, memory, or hit rate limits on external services (e.g., databases, APIs).

Careful monitoring of worker resource utilization and queue depth is essential to tune concurrency settings. Auto-scaling worker groups can dynamically adjust the number of worker instances based on queue length or CPU usage.

Monitoring & Observability

Just like any other service, workers need robust observability.

- **Logging:** Detailed logs about task start, progress, completion, and errors. Include correlation IDs to trace a task end-to-end.
- **Metrics:**
    - Queue depth (number of messages waiting).
    - Task processing rate (tasks/second).
    - Task processing time (latency).
    - Number of retries.
    - Error rates.
    - Worker resource utilization (CPU, memory).
- **Tracing:** Distributed tracing (as discussed in Chapter 5) is invaluable for understanding the full lifecycle of a task as it moves from producer to queue to worker and potentially other services.

Worker Patterns in AI/Agentic Systems

AI and agentic systems are prime candidates for worker architectures because their tasks are often:

- **Long-running:** Training models, generating images, complex data analysis, multi-step agentic workflows.
- **Resource-intensive:** Require significant CPU, GPU, or memory.
- **Asynchronous by nature:** Users don't expect instant results for complex AI operations.

Real-world scenario: An AI Agent Workflow

Consider an AI agent designed to generate a personalized marketing campaign for a user. This might involve several steps:

1. **Data Retrieval (Producer):** A user interaction triggers the campaign generation. An orchestrator service (producer) creates a task for the AI agent, including the user ID and campaign parameters.
2. **Queueing:** This task is placed on a "Campaign Generation" message queue.
3. **Agent Worker (Consumer):** An AI agent worker picks up the task.
    - **Step 1: User Profile Enrichment:** The worker calls a profile service to fetch detailed user data.
    - **Step 2: Content Generation:** The worker uses a Large Language Model (LLM) to draft personalized email content, social media posts, and ad copy. This can be a very long process.
    - **Step 3: Image Generation:** The worker interacts with an image generation model to create custom visuals.
    - **Step 4: A/B Test Variant Creation:** The worker generates multiple variants for testing.
    - **Step 5: Campaign Assembly:** The worker combines all generated content and schedules it with a marketing automation platform.
4. **Status Updates:** Throughout this process, the worker periodically updates the status of the campaign generation in a database, allowing the user to track progress.
5. **Completion Notification:** Once complete, the worker might put a "Campaign Ready" message on another queue, triggering a notification service to inform the user.

In this scenario, the AI agent itself acts as a sophisticated worker, orchestrating multiple sub-tasks. Each sub-task could even be offloaded to another set of specialized workers (e.g., a dedicated image generation worker pool). This layered approach demonstrates the power of composable worker architectures.

Step-by-Step Walkthrough: Illustrating an Image Processing Worker's Logic

Let's walk through the conceptual steps an engineer would take to implement an asynchronous image processing worker. We'll use Python-like pseudocode to illustrate the logical flow, rather than providing a runnable production setup. This helps us focus on the core interactions and responsibilities.

Scenario: A web application allows users to upload profile pictures. After upload, images need to be resized to several standard dimensions, watermarked, and stored in a cloud storage bucket.

1. **Identify the Asynchronous Task:**
    - **Problem:** Image resizing and watermarking are CPU-bound and can take time, blocking the user's upload request.
    - **Solution:** Offload these operations to a worker.

2. **Design the Message (Task Payload):** The message sent to the queue needs to contain all the necessary information for the worker to perform its job. We'll typically use a structured format like JSON.

```
{
    "task_id": "unique-uuid-for-this-task-123",
    "image_id": "uploaded-image-uuid-abc",
    "source_url": "https://temp-storage.example.com/uploaded-image-uuid-
abc.jpg",
    "user_id": "user-uuid-xyz",
    "target_sizes": [
        {"width": 100, "height": 100},
        {"width": 400, "height": 400}
    ],
    "watermark_text": "MyCompany"
}
```

   - `task_id`: A unique ID for this specific processing task. Crucial for idempotency.
   - `image_id`: Unique ID for the uploaded image itself.
   - `source_url`: The temporary location of the original uploaded image. We pass a URL to avoid putting large binary data directly into the queue.
   - `user_id`: To associate the processed image with a user.
   - `target_sizes`: A list of dimensions for the resized images.
   - `watermark_text`: Optional text to add as a watermark.

3. **Implement the Producer (Web Server Logic):** This is the part of your web application that receives the image upload and publishes the task.

```
# web_server_app.py (Illustrative Python-like pseudocode)

import uuid
import json
# Assume 'queue_client' is an initialized client for your message queue
(e.g., SQS, RabbitMQ)
# Assume 'storage_client' handles saving files to temporary storage
(e.g., S3, Azure Blob)

def handle_image_upload(request):
    # 1. Receive image data from user request
    uploaded_file = request.files['profile_picture']

    # 2. Generate unique IDs
    image_id = str(uuid.uuid4())
    task_id = str(uuid.uuid4()) # A new ID for the processing task

    # 3. Store the original image in a temporary location
    # This function would upload the file and return its accessible URL
    source_url = storage_client.upload_temp_image(uploaded_file,
image_id)

    # 4. Construct the task message
    task_payload = {
        "task_id": task_id,
        "image_id": image_id,
        "source_url": source_url,
        "user_id": request.user.id,
        "target_sizes": [{"width": 100, "height": 100}, {"width": 400, "h
eight": 400}],
        "watermark_text": "MyCompanyLogo"
    }

    # 5. Publish the message to the "image-processing-queue"
    queue_client.send_message("image-processing-queue", json.dumps(task_p
ayload))

    # 6. Immediately return a 202 Accepted response
    # This tells the user the request was received and is being processed
    return {
        "status": "processing",
        "message": "Image upload received, processing in background.",
        "image_id": image_id,
        "status_url": f"/api/image-status/{image_id}" # URL for user to
check progress
    }, 202
```

- **`import uuid, json`**: Standard libraries for unique identifiers and structured data.
- **`storage_client.upload_temp_image(...)`**: This represents saving the raw image file to cloud storage and getting a URL. This prevents large files from clogging the queue.

- `task_id = str(uuid.uuid4())`: A unique identifier for this specific processing job. If the message is redelivered, this ID helps the worker know if it's already processed it.
- `queue_client.send_message(...)`: Sends the JSON payload to our designated message queue.
- `return ..., 202`: The `202 Accepted` HTTP status code is a standard way to indicate that a request has been accepted for processing, but the processing is not yet complete. This provides immediate feedback to the user.

4. **Set Up the Message Queue:** You would configure your chosen message queue service (e.g., AWS SQS, RabbitMQ) to create the `image-processing-queue`. Crucially, you would also set up a **Dead-Letter Queue (DLQ)**. This DLQ will automatically receive messages that fail to be processed successfully after a specified number of retries, preventing "poison messages" from endlessly blocking your main queue.

5. **Develop the Worker Service Logic:** This is the core logic that runs on your worker instances. It continuously pulls messages from the queue and performs the image processing.

```
# image_worker_service.py (Illustrative Python-like pseudocode)

import json
# Assume 'queue_client' is an initialized client for your message queue
# Assume 'storage_client' handles downloading/uploading images to cloud
storage
# Assume 'image_processor' contains functions for resizing, watermarking
# Assume 'db_client' for updating image status in a database

def process_image_task(task_payload):
    task_id = task_payload["task_id"]
    image_id = task_payload["image_id"]
    source_url = task_payload["source_url"]
    target_sizes = task_payload["target_sizes"]
    watermark_text = task_payload["watermark_text"]

    # 1. Idempotency Check: Has this specific task_id already been
processed?
    if db_client.get_task_status(task_id) == "completed":
        print(f"Task {task_id} already completed, skipping.")
        return # Exit early if already done

    # 2. Mark task as 'in_progress'
    db_client.update_task_status(task_id, "in_progress")
    db_client.update_image_status(image_id, "processing")

    try:
        # 3. Download the original image
        original_image_data = storage_client.download_image(source_url)
        processed_image_urls = []

        # 4. Iterate through target sizes, resize, watermark, and upload
        for size_config in target_sizes:
            width, height = size_config["width"], size_config["height"]
            print(f"Processing {image_id} to {width}x{height}")

            # Resize and watermark
            resized_image = image_processor.resize(original_image_data, w
idth, height)
            if watermark_text:
                resized_image = image_processor.apply_watermark(resized_i
mage, watermark_text)

            # Upload processed image to permanent storage
            processed_url = storage_client.upload_processed_image(
                resized_image, image_id, f"{width}x{height}.jpg"
            )
            processed_image_urls.append(processed_url)

        # 5. Update database with processed image URLs and mark as
complete
        db_client.update_image_data(image_id, {"processed_urls": processe
d_image_urls})
        db_client.update_task_status(task_id, "completed")
        db_client.update_image_status(image_id, "ready")
```

```
        print(f"Successfully processed image {image_id} for task
{task_id}")

    except Exception as e:
        # 6. Error Handling: Log the error and mark task as failed
        print(f"Error processing task {task_id} for image {image_id}:
{e}")
        db_client.update_task_status(task_id, "failed",
error_message=str(e))
        db_client.update_image_status(image_id, "failed")

# Re-raise the exception to signal to the queue that this message needs
to be retried
        raise

# Main worker loop
def start_worker():
    print("Worker started, waiting for messages...")
    while True:
        # 7. Poll for messages from the queue
        message = queue_client.receive_message("image-processing-queue")
        if message:
            try:
                task_payload = json.loads(message.body)
                process_image_task(task_payload)
                # 8. Acknowledge message after successful processing
                queue_client.delete_message("image-processing-queue", mes
sage.receipt_handle)
            except Exception as e:
                print(f"Failed to process message or task: {e}. Message
will be retried or moved to DLQ.")
                # The queue client implicitly handles retries/DLQ if
message is not deleted
        else:
            # No messages, wait a bit before polling again
            time.sleep(5)
```

- **`db_client.get_task_status(task_id)`**: This is our idempotency check. Before doing any work, the worker checks if this specific `task_id` has already been successfully processed. This prevents duplicate work if the message is redelivered.
- **`db_client.update_task_status(...)`**: Updates a database record for the task's state. This is crucial for allowing the user to check the progress via the `status_url` returned by the producer.
- **`storage_client.download_image(...)`**: Fetches the original image from the temporary storage.
- **`image_processor.resize(...)`**, **`image_processor.apply_watermark(...)`**: These represent the actual image manipulation logic.
- **`storage_client.upload_processed_image(...)`**: Saves the final processed images to a permanent, accessible location.

- `try...except...raise`: This robust error handling ensures that if any step within the processing fails, the task status is updated, and the exception is re-raised. This signals to the queue system that the message was not successfully processed, allowing it to be retried or moved to the DLQ.
- `queue_client.receive_message(...)`: Continuously fetches messages from the queue.
- `queue_client.delete_message(...)`: This is the **acknowledgment** step. Only after the entire task has been successfully processed and its state persisted should the message be deleted from the queue. If the worker crashes before this, the message will eventually become visible again for another worker to pick up.

6. **Deployment and Scaling:** Once the producer and worker logic are ready, you would deploy multiple instances of your worker service to handle the load. Cloud platforms offer managed services (like AWS ECS, Kubernetes, Azure Container Apps) that can automatically scale the number of worker instances up or down based on metrics like queue depth (how many messages are waiting) or worker CPU utilization.
7. **Monitoring and Alerting:** The final piece is robust observability. You'd set up dashboards to visualize:
    - The **queue depth** of `image-processing-queue`.
    - The **rate** at which messages are being produced and consumed.
    - The **latency** (how long tasks sit in the queue before processing).
    - The **CPU and memory utilization** of your worker instances.
    - The number of messages in the **Dead-Letter Queue**. Alerts would notify your team if any of these metrics cross predefined thresholds, indicating a potential bottleneck or failure.

This conceptual walkthrough highlights the thought process involved in designing a robust worker system, emphasizing decoupling, error handling, and scalability, with concrete (though illustrative) logic examples.

Mini-Challenge: Designing an AI Agent Email Campaign Worker

Your turn! Imagine you're building a system where an AI agent generates personalized marketing emails. This task involves several steps: fetching user data, calling an LLM for content, potentially generating images, and then scheduling the email.

Challenge: Outline the architecture for an "AI Email Campaign Worker" system. Think about:

1. **Producer:** What service would initiate this task, and what information would it put into the message?
2. **Message Queue:** What kind of queue would you use, and why?
3. **Worker Service:** What are the key internal steps the AI agent worker would perform?
4. **Error Handling:** How would you make sure a failing email generation doesn't block the entire system?
5. **Idempotency:** What unique identifier would you use to ensure a campaign isn't accidentally generated twice?

Hint: Consider the "AI Agent Workflow" example we just discussed. Focus on the data flow and responsibilities of each component.

Common Pitfalls & Troubleshooting

Even with careful design, worker architectures introduce new complexities. Here are some common pitfalls:

- **Over-engineering:** `⚠️ What can go wrong:` Don't use a worker system for tasks that are inherently fast and synchronous. Adding a queue and worker for a 50ms operation introduces unnecessary latency, complexity, and operational overhead. `🔥 Optimization / Pro tip:` Evaluate the task's latency, resource consumption, and failure impact. If it's fast, reliable, and non-blocking, keep it synchronous.

- **Message Too Large:** ⚠️ `What can go wrong:` Putting large binary data (like raw images or videos) directly into message queues can significantly slow down the queue, increase costs, and potentially exceed message size limits. 🔥 `Optimization / Pro tip:` Store large payloads in dedicated storage (e.g., S3, Azure Blob, GCS) and pass only the reference (URL or ID) in the message.
- **Lost Messages / Lack of Acknowledgment:** ⚠️ `What can go wrong:` If a worker crashes before acknowledging a message, but after processing it, the message might be redelivered, leading to duplicate processing if not idempotent. Conversely, if it acknowledges too early and then fails, the task is lost. 🔥 `Optimization / Pro tip:` Implement "at-least-once" delivery semantics by acknowledging messages only after the task is fully and successfully completed (and its state persisted). Rely on idempotency to handle potential duplicates.
- **Worker Starvation or Overload:** ⚠️ `What can go wrong:` If the rate of incoming messages exceeds the processing capacity of your workers, the queue depth will grow indefinitely, leading to massive delays. Conversely, too many workers can exhaust shared resources (like database connections). 🔥 `Optimization / Pro tip:` Implement auto-scaling for workers based on queue depth and worker resource utilization. Monitor these metrics closely.
- **Idempotency Failures:** ⚠️ `What can go wrong:` Assuming an operation is idempotent when it's not, or failing to properly implement idempotency checks, can lead to data corruption or unintended side effects when messages are redelivered. 🔥 `Optimization / Pro tip:` Rigorously test your worker's idempotency under failure conditions. Use unique transaction IDs and atomic operations to ensure consistency.

Summary

Worker architectures are a cornerstone of modern, scalable, and resilient distributed systems. By embracing asynchronous processing, you can decouple long-running tasks from your core application, leading to a more responsive user experience and efficient resource management.

Here are the key takeaways:

- **Decoupling:** Workers separate task initiation from execution, preventing synchronous blocking.

- **Core Components:** Task Producers, Message Queues (like Kafka, RabbitMQ, SQS), and Worker Services are essential.
- **Robust Design:** Idempotency, comprehensive error handling with retries and Dead-Letter Queues, and careful concurrency management are critical for reliability.
- **Observability:** Logging, metrics, and tracing are vital for monitoring worker health and task progress.
- **AI Integration:** Worker patterns are naturally suited for the computationally intensive and asynchronous nature of AI/agentic workflows.
- **Avoid Over-engineering:** Only apply worker architectures when the benefits (scalability, responsiveness, resilience) outweigh the added complexity for genuinely long-running tasks.

In the next chapter, we'll expand on asynchronous processing by exploring **Event-Driven Systems**, where services communicate not just by passing tasks, but by emitting and reacting to significant events, enabling even greater flexibility and scalability.

References

- Microservices Architecture Style - Azure Architecture Center
- Apache Kafka Documentation
- RabbitMQ Documentation
- Amazon SQS Developer Guide
- Azure Service Bus Documentation
- Martin Fowler - Idempotent Operations

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 07

Event-Driven Architectures: Building Reactive and Scalable Systems

Introduction: Embracing Reactivity for Modern Systems

Imagine a bustling city where every action immediately triggers a cascade of necessary responses without anyone having to wait. A taxi drops off a passenger, and immediately, its status updates, a new fare is assigned, and a billing record is created. This highly responsive, interconnected flow is the essence of an event-driven architecture (EDA). It's how complex systems stay agile and responsive, even under immense load.

In this chapter, we'll dive deep into Event-Driven Architectures, a paradigm that enables systems to react to changes as they happen, fostering incredible scalability, resilience, and responsiveness. We'll explore the fundamental concepts, key components, and practical patterns that empower applications to evolve from tightly coupled monoliths into agile, reactive ecosystems.

This journey builds upon our previous discussions on service-to-service communication and asynchronous workflows. While we've touched upon queues as a mechanism for decoupling, EDA takes this concept to its logical conclusion, making events the primary means of communication and state change across your entire system. Get ready to think about your applications as living, breathing entities constantly reacting to the world around them.

The Core Idea: Events as the Language of Change

At its heart, an event-driven architecture revolves around events. An event is simply a record of something that has happened. It's a fact, an immutable statement about a change of state in your system.

What Exactly is an Event?

Think of an event like a newspaper headline: "User Registered," "Order Placed," "Payment Processed." These are concrete, past-tense statements. An event doesn't tell a system what to do; it simply announces what did happen. Other parts of the system can then decide if they care about that particular event and react accordingly.

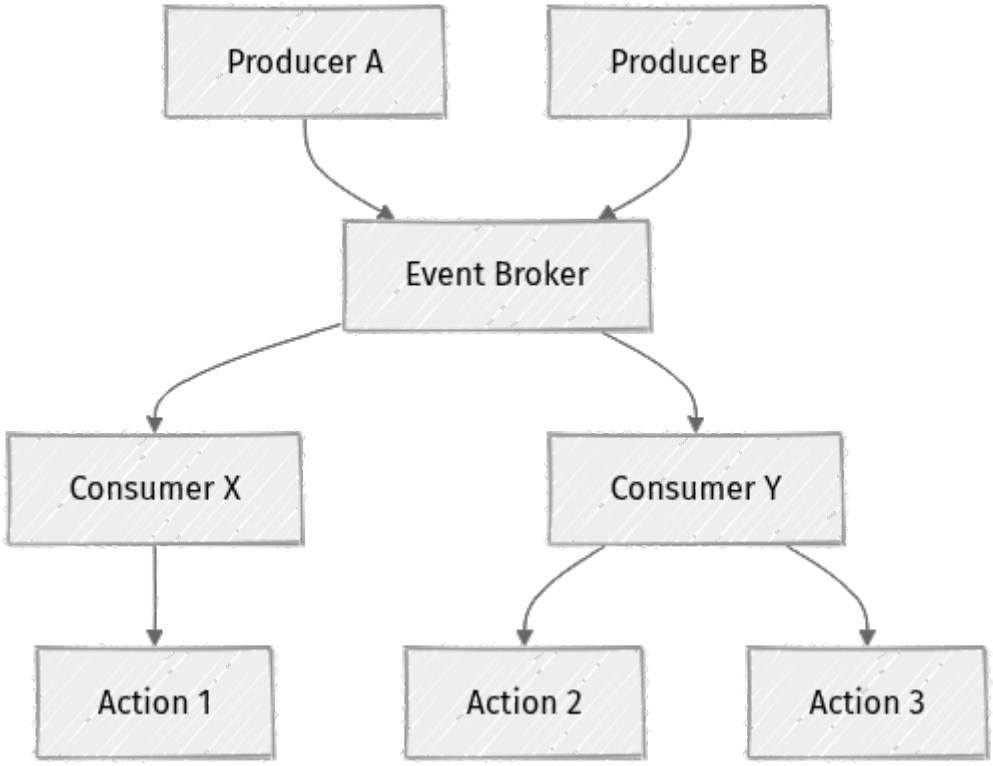
Why does this matter? This simple shift from direct commands to event notifications fundamentally changes how components interact. Instead of Service A directly calling Service B and waiting for a response (which creates tight coupling), Service A simply announces "X happened." Service B (and C, D, etc.) can then independently listen for "X happened" and take action. This significantly reduces dependencies.

Producers, Brokers, and Consumers: The Event Ecosystem

An event-driven system typically consists of three main roles:

1. **Event Producers (Publishers):** These are the services or components that detect a change in their state and publish an event to notify the rest of the system. They don't know or care who listens; they just publish the fact.
2. **Event Brokers (Message Brokers):** This is the central nervous system of an EDA. A broker is responsible for receiving events from producers and reliably delivering them to interested consumers. It acts as a buffer and a router, ensuring events aren't lost and reach their intended destinations.
3. **Event Consumers (Subscribers):** These are services or components that express interest in specific types of events. When an event they care about arrives at the broker, the consumer processes it, potentially updating its own state or triggering further actions (which might, in turn, publish new events).

This model fosters extreme **decoupling**. Producers don't need to know about consumers, and consumers don't need to know about producers. They only need to agree on the format of the events. This makes systems more flexible, easier to scale, and more resilient to individual component failures.



A simple event-driven flow showing multiple producers sending events to a broker, which then distributes them to various consumers for processing.

Key Components in Practice: The Event Broker

The event broker is arguably the most critical component in an EDA. It's not just a pass-through; it provides crucial guarantees for reliability and scalability.

What an Event Broker Does

An event broker acts as a middleware that facilitates asynchronous communication between services. Its core responsibilities include:

- **Buffering:** Storing events temporarily if consumers are slow or unavailable.
- **Routing:** Directing events to the correct consumers or groups of consumers.
- **Durability:** Ensuring events are not lost, even if the broker or consuming services crash.
- **Ordering (Optional but Important):** For some event streams, maintaining the order in which events were published is critical.
- **Fan-out:** Allowing multiple consumers to receive the same event, enabling parallel processing or different reactions to a single event.

Why is it crucial? Without a robust broker, producers would need to manage consumer lists, retry logic, and error handling themselves, leading to tightly coupled and fragile systems. The broker abstracts away these complexities, providing a centralized, reliable backbone.

Types of Brokers: Queues vs. Topics

While the terms "message broker" and "event broker" are often used interchangeably, it's helpful to distinguish between two primary modes of operation:

1. **Queues (Point-to-Point Messaging):**
    - **Concept:** A queue delivers each message to exactly one consumer. If multiple consumers listen to the same queue, they effectively share the workload, distributing messages among themselves.
    - **Use Case:** Task distribution, load balancing, ensuring a single worker processes a specific job. Think of a customer support queue where each new ticket is handled by one available agent.
    - **Examples:** AWS SQS, Azure Service Bus Queues, RabbitMQ queues.
2. **Topics (Publish-Subscribe Messaging):**
    - **Concept:** A topic delivers each message to all interested subscribers. A single event published to a topic can be processed by many different services simultaneously.
    - **Use Case:** Broadcasting events where multiple independent services need to react. Think of a "New Order" event that needs to trigger payment processing, inventory updates, and a notification email, all at once.
    - **Examples:** Apache Kafka, AWS SNS, Azure Service Bus Topics, RabbitMQ exchanges with fan-out.

⚡ **Quick Note:** Modern brokers like Apache Kafka (version 3.7.0 as of 2026-05-15) often combine aspects of both, offering durable, ordered streams that can be consumed by multiple groups of consumers, effectively acting as both a queue and a topic depending on how consumers are configured. For cloud-native options, AWS SQS (queues) and SNS (topics) or Azure Service Bus (both queues and topics) are popular choices.

Choosing the Right Broker: Trade-offs

The choice of event broker depends on your specific needs:

- **Throughput & Latency:** Kafka is renowned for high throughput and low-latency streaming, handling millions of events per second.
- **Message Guarantees:** Do you need "at-most-once," "at-least-once," or "effectively once" delivery semantics? Remember, "exactly-once" is exceptionally difficult to achieve perfectly in distributed systems and often implies "effectively once" through idempotent consumers.
- **Durability:** How long do you need events to persist? Days, weeks, or indefinitely?
- **Complexity:** Managed cloud services (SQS/SNS, Azure Service Bus) offer simplicity, while self-managed Kafka requires more operational expertise and infrastructure management.

Event Consumers: Building Resilient Reactions

Consumers are the actors that bring your event-driven system to life. They listen, react, and often produce new events.

Idempotency: The Golden Rule for Consumers

In distributed systems, it's almost guaranteed that a consumer might receive the same event more than once due to network retries, broker re-delivery, or consumer restarts. This is where **idempotency** becomes critical.

📌 **Key Idea:** An operation is idempotent if executing it multiple times produces the same result as executing it once.

For example, if an `UpdateUserBalance` event increases a user's balance by $10, simply adding $10 every time the event is received is not idempotent. Instead, the consumer should check if that specific balance update has already been applied using a unique transaction ID associated with the event. If it has, it simply acknowledges the event without re-processing. This prevents double-counting or erroneous state changes.

Error Handling and Dead-Letter Queues (DLQs)

What happens if a consumer fails to process an event?

- **Retries:** Most brokers offer automatic retry mechanisms. If a consumer fails to process an event (e.g., due to a temporary database issue), the event might be re-delivered a few times after a delay.

- **Dead-Letter Queues (DLQs):** If an event consistently fails after several retries, it's typically moved to a DLQ. This prevents "poison pill" messages (events that always cause a consumer to crash) from blocking the main queue and allows engineers to inspect and manually reprocess or discard problematic events. DLQs are a critical part of robust asynchronous processing and debugging.

Step-by-Step: Implementing an Event-Driven Order Flow

Let's walk through a conceptual implementation of an event-driven flow for a classic e-commerce scenario: processing a customer order. We'll use a simplified Python-like pseudocode to illustrate the core logic without getting bogged down in specific broker client libraries.

Scenario: A customer places an order. Multiple independent actions need to happen: payment processing, inventory reservation, and shipping preparation.

Step 1: The Order Service (Producer) Publishes an Event

The `Order Service` is responsible for receiving the initial order request, persisting it, and then announcing that an order has been placed.

```
# order_service.py

import json
import uuid
# Assume an event_broker_client is available for publishing
from event_broker import publish_event

def place_order(customer_id, items):
    order_id = str(uuid.uuid4())
    # 1. Persist order details in Order Service's database
    #    (e.g., save_order_to_db(order_id, customer_id, items,
status="PENDING"))
    print(f"Order {order_id} received and saved as PENDING.")

    # 2. Construct the OrderPlaced event
    event_data = {
        "order_id": order_id,
        "customer_id": customer_id,
        "items": items,
        "timestamp": "2026-05-15T10:00:00Z", # Current timestamp
        "event_id": str(uuid.uuid4()) # Unique ID for this specific event
instance
    }
    event = {
        "type": "OrderPlaced",
        "payload": event_data
    }

    # 3. Publish the event to the 'orders' topic
    publish_event("orders", json.dumps(event))
    print(f"Published OrderPlaced event for order {order_id}")
```

```
    # 4. Return immediate confirmation to the customer
    return {"message": "Order received, processing initiated.", "order_id": or
der_id}

# Example usage:
# place_order("user123", [{"product_id": "A", "quantity": 1}, {"product_id":
"B", "quantity": 2}])
```

Explanation:

- The `order_service.py` function `place_order` first saves the order (conceptually) to its local database with a `PENDING` status.
- It then creates an `OrderPlaced` event, including a unique `event_id` for idempotency tracking.
- Finally, it uses `publish_event` to send this event to an `orders` topic on the event broker. The `Order Service` doesn't know or care who will process this event; it just announces the fact.
- A quick confirmation is returned to the customer, making the system feel responsive.

Step 2: The Payment Service (Consumer & Producer) Reacts

The `Payment Service` listens for `OrderPlaced` events. When it receives one, it attempts to process the payment.

```
# payment_service.py

import json
# Assume an event_broker_client for consuming and publishing
from event_broker import subscribe_to_topic, publish_event

def process_payment_for_order(event_data):
    order_id = event_data["order_id"]
    customer_id = event_data["customer_id"]
    items = event_data["items"]
    event_id = event_data["event_id"] # For idempotency

    # 1. Check for idempotency (crucial!)
    #    (e.g., if is_event_already_processed("PaymentService", event_id):
return)
    print(f"Payment Service received OrderPlaced for order {order_id}.
Processing payment...")

    # Simulate payment processing logic
    payment_successful = True # In a real system, this involves external APIs
    if payment_successful:
        print(f"Payment successful for order {order_id}.")
        # 2. Publish PaymentProcessed event
        payment_event = {
            "type": "PaymentProcessed",
            "payload": {
                "order_id": order_id,
                "customer_id": customer_id,
```

```
                "transaction_id": str(uuid.uuid4()),
                "timestamp": "2026-05-15T10:01:00Z",
                "event_id": str(uuid.uuid4())
            }
        }
        publish_event("payments", json.dumps(payment_event))
    else:
        print(f"Payment failed for order {order_id}.")
        # 3. Publish PaymentFailed event
        failure_event = {
            "type": "PaymentFailed",
            "payload": {
                "order_id": order_id,
                "customer_id": customer_id,
                "reason": "Insufficient funds",
                "timestamp": "2026-05-15T10:01:00Z",
                "event_id": str(uuid.uuid4())
            }
        }
        publish_event("payments", json.dumps(failure_event))

# To start the consumer:
# subscribe_to_topic("orders", process_payment_for_order)
```

Explanation:

- The `Payment Service` subscribes to the `orders` topic.
- When an `OrderPlaced` event arrives, it first performs an idempotency check to ensure it doesn't process the same event twice.
- It then simulates payment processing.
- Based on the outcome, it publishes either a `PaymentProcessed` or `PaymentFailed` event to a `payments` topic. This is a classic example of a service acting as both a consumer and a producer.

Step 3: The Inventory Service (Consumer) Reacts to Payment

The `Inventory Service` only cares if a payment was successful before attempting to reserve stock.

```
# inventory_service.py

import json
from event_broker import subscribe_to_topic, publish_event

def reserve_inventory_for_order(event_data):
    order_id = event_data["order_id"]
    items = event_data["items"]
    event_id = event_data["event_id"] # For idempotency

    # 1. Check for idempotency
    #    (e.g., if is_event_already_processed("InventoryService", event_id):
return)
    print(f"Inventory Service received PaymentProcessed for order {order_id}.
Reserving items...")
```

```
    # Simulate inventory reservation logic
    inventory_available = True # In a real system, check stock levels
    if inventory_available:
        print(f"Inventory reserved for order {order_id}.")
        # 2. Publish InventoryReserved event
        inventory_event = {
            "type": "InventoryReserved",
            "payload": {
                "order_id": order_id,
                "items": items,
                "timestamp": "2026-05-15T10:02:00Z",
                "event_id": str(uuid.uuid4())
            }
        }
        publish_event("inventory", json.dumps(inventory_event))
    else:
        print(f"Inventory insufficient for order {order_id}.")
        # 3. Publish InventoryFailed event (triggering refund)
        failure_event = {
            "type": "InventoryFailed",
            "payload": {
                "order_id": order_id,
                "items": items,
                "reason": "Out of stock",
                "timestamp": "2026-05-15T10:02:00Z",
                "event_id": str(uuid.uuid4())
            }
        }
        publish_event("inventory", json.dumps(failure_event))

# To start the consumer:
# subscribe_to_topic("payments", reserve_inventory_for_order)
```

Explanation:

- The `Inventory Service` subscribes to the `payments` topic, specifically looking for `PaymentProcessed` events.
- It performs its idempotency check.
- It then attempts to reserve inventory. If successful, it publishes an `InventoryReserved` event; if not, an `InventoryFailed` event.

Step 4: The Shipping Service (Consumer) Reacts to Inventory Reservation

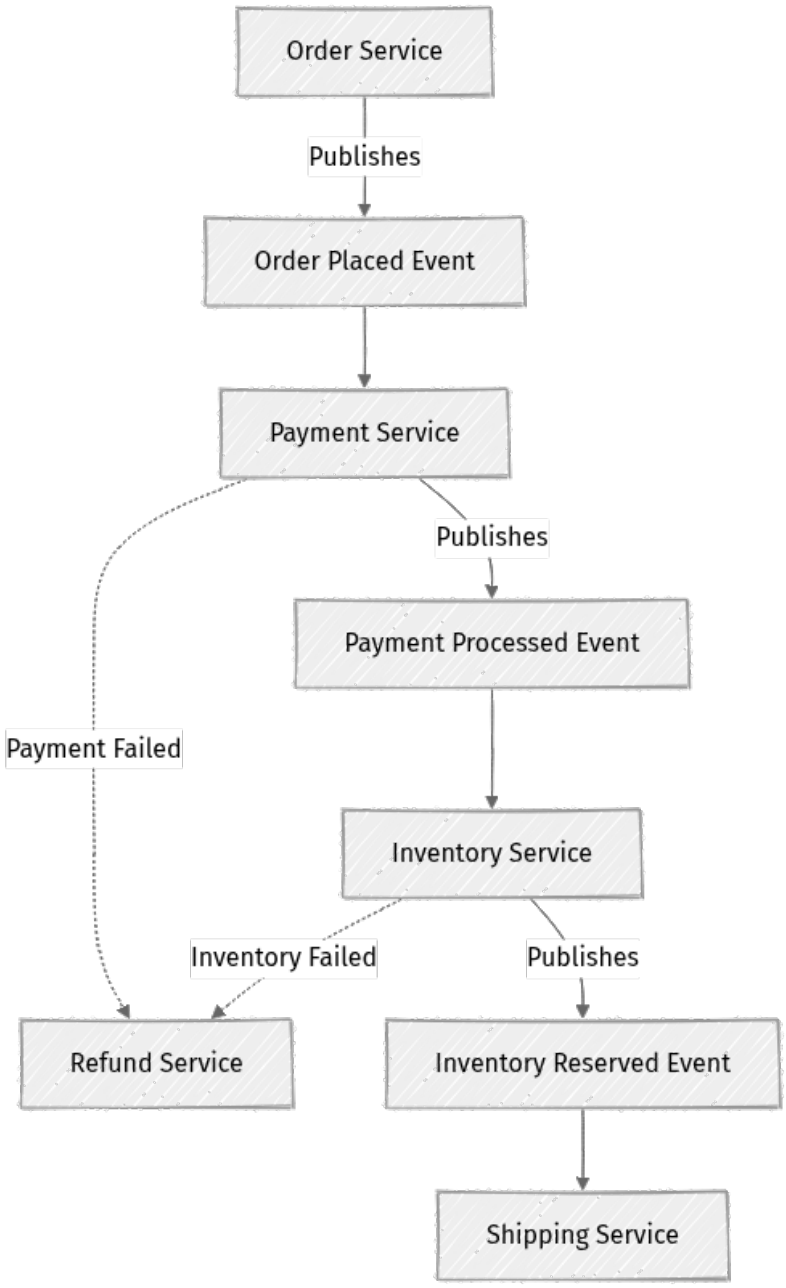
Finally, the `Shipping Service` initiates shipping once inventory is confirmed.

```
# shipping_service.py

import json
from event_broker import subscribe_to_topic

def prepare_shipment(event_data):
    order_id = event_data["order_id"]
    items = event_data["items"]
    event_id = event_data["event_id"] # For idempotency
```

```
    # 1. Check for idempotency
    #    (e.g., if is_event_already_processed("ShippingService", event_id):
return)
    print(f"Shipping Service received InventoryReserved for order {order_id}.
Preparing shipment...")
    # Simulate shipment preparation (e.g., generate shipping label, notify
warehouse)
    print(f"Shipment for order {order_id} is being prepared.")

# To start the consumer:
# subscribe_to_topic("inventory", prepare_shipment)
```

Explanation:

- The `Shipping Service` subscribes to the `inventory` topic, specifically `InventoryReserved` events.
- Upon receiving the event, it performs an idempotency check and then initiates the shipping process.



An event-driven order processing flow, demonstrating how services react to specific events published to a central broker.

This step-by-step approach demonstrates the power of EDA: the `Order Service` doesn't need to know anything about payments, inventory, or shipping; it just announces that an order was placed. This allows each service to scale and evolve independently, making the system much more robust and flexible.

Event-Driven Design Patterns

Beyond the basic publish-subscribe, several powerful patterns emerge in EDA.

Event Sourcing: The Ledger of Changes

What is it? Instead of storing only the current state of an entity (e.g., a `User` record with current balance), Event Sourcing stores every change to that entity as a sequence of immutable events. The current state is then derived by replaying these events.

Why does it exist?

- **Auditability:** Provides a complete, unalterable history of everything that ever happened to an entity.
- **Temporal Queries:** You can easily reconstruct the state of an entity at any point in time.
- **Debugging:** Understanding how an entity reached a particular (potentially erroneous) state is much simpler.
- **Decoupling:** Different services can subscribe to the event stream to build their own read models optimized for their needs.

⚠️ **What can go wrong:** Event Sourcing adds complexity. Replaying a long history of events to get the current state can be slow, requiring "snapshots" at intervals. Schema evolution of events over time also needs careful management.

CQRS (Command Query Responsibility Segregation)

What is it? CQRS separates the model used to update data (the "command" model) from the model used to read data (the "query" model).

How it relates to EDA: In an EDA, the command model might publish events after processing a command (e.g., `OrderCreated`). These events can then be used to update one or more separate, optimized read models (e.g., a denormalized view for a customer dashboard, a search index).

Benefits:

- **Scalability:** Read and write models can be scaled independently, which is crucial for systems with high read-to-write ratios.
- **Optimized Performance:** Read models can be highly optimized for queries (e.g., using a NoSQL database for fast lookups, or a search engine).
- **Flexibility:** Different services can have their own read models tailored to their specific querying needs.

When to use/not use: CQRS is powerful but introduces significant complexity. It's best suited for domains where read and write patterns are very different, or where extreme read scalability is required. For simpler applications, the overhead often outweighs the benefits.

Sagas: Managing Distributed Transactions

What is it? A saga is a sequence of local transactions, where each transaction updates a service's own database and publishes an event to trigger the next step in the saga. If a step fails, compensating transactions are executed to undo the changes made by previous steps.

Orchestration vs. Choreography:

- **Choreography:** Each service publishes an event, and other services react to it (like our e-commerce example). This is highly decoupled but can be hard to trace the overall flow, especially in complex scenarios.
- **Orchestration:** A central "saga orchestrator" service manages the sequence of steps, sending commands to participants and reacting to their responses (often via events). This provides clearer flow control but introduces a central point of coordination.

Why is it important? Distributed transactions (ACID properties across multiple services) are notoriously difficult. Sagas provide a pattern to achieve eventual consistency across services without relying on two-phase commits, which are often impractical in microservice architectures.

AI Agents and Event-Driven Systems

The rise of AI agents makes event-driven architectures even more compelling. Agents, by their nature, are designed to perceive, reason, and act – making them perfect candidates for event consumption and production.

How AI Agents Leverage Events

- **Reactive Intelligence:** AI agents can subscribe to event streams to gain real-time awareness of system changes. For instance, a "Fraud Detection Agent" might subscribe to `PaymentProcessed` events to immediately evaluate transactions for suspicious activity.
- **Proactive Actions:** After processing information, an AI agent can publish new events to trigger subsequent actions in the system. An "Inventory Optimization Agent" might publish an `InventoryRestockRecommended` event after analyzing sales trends and stock levels.
- **Multi-Agent Orchestration:** Complex AI workflows involving multiple agents (e.g., a "Customer Service Agent" escalating to a "Technical Support Agent") can be orchestrated efficiently using events as the communication backbone. Agent A publishes `IssueEscalated`, and Agent B consumes it.

⚡ **Real-world insight:** As of 2026, many AI platforms and agent frameworks are designed with eventing in mind, using internal message buses or integrating with external brokers. This allows for modular, scalable AI components that can operate asynchronously and react to dynamic environments, often processing millions of events per hour.

Example: An AI-Driven Fraud Detection Agent

Consider an AI agent specifically designed to detect fraudulent payments:

1. **Consumes:** `PaymentProcessed` events from the Event Broker. Each event contains transaction details (amount, user, location, etc.).
2. **Reasons:** The agent's model analyzes these details in real-time, perhaps cross-referencing with historical fraud data, user behavior patterns, or external risk scores. This often involves real-time inference against pre-trained models.
3. **Acts:**
    - If the transaction is deemed high-risk, the agent publishes a `FraudDetected` event.
    - If the transaction is low-risk, it might publish a `PaymentApprovedByFraudAgent` event (or simply do nothing, allowing the default flow to proceed).

Other services (e.g., a `CustomerNotification Service` or a `PaymentBlocking Service`) can then subscribe to `FraudDetected` events to take appropriate action. This demonstrates how AI agents become first-class citizens in an event-driven ecosystem, enhancing system capabilities without introducing tight coupling.

Mini-Challenge: Design an Event Flow

Let's put your understanding to the test!

Challenge: Imagine you're building a new social media platform. Design an event-driven flow for what happens when a **new user signs up**. Think about at least three distinct actions that different services might need to take in response.

- **Task:**
    1. Identify the initial event.
    2. List at least three services that would consume this event.
    3. For each consuming service, describe its action and any new events it might publish.
    4. Draw a simple Mermaid `flowchart` diagram illustrating your design.
- **Hint:** Consider actions like sending a welcome email, initializing a user profile in a separate service, or adding the user to an analytics system. Remember to keep services decoupled!
- **What to observe/learn:** This exercise highlights how a single event can fan out to trigger multiple parallel, independent processes, improving responsiveness and system design.

Common Pitfalls and Trade-offs

While powerful, Event-Driven Architectures are not a silver bullet. Understanding their complexities and trade-offs is crucial.

1. Increased Complexity and Distributed Debugging

- **Problem:** The asynchronous nature and decoupling can make it harder to trace the end-to-end flow of a request. When a problem occurs, it's not always obvious which service or event caused it. This "spaghetti of events" can be a nightmare without proper tools.
- **Mitigation:** Robust observability (logging, metrics, and especially distributed tracing, which we'll cover in a later chapter) is paramount. Each event should carry a correlation ID to link related operations across services, allowing you to follow a single transaction across multiple hops.

2. Event Schema Management

- **Problem:** Events are contracts between producers and consumers. As your system evolves, event schemas will change. How do you handle old consumers with new event versions, or new consumers with old event versions? Breaking changes can cause cascading failures.

- **Mitigation:** Implement strict schema versioning (e.g., `OrderPlaced_v1`, `OrderPlaced_v2`). Favor backward-compatible changes (adding optional fields) over breaking ones. Employ schema registries (like Confluent Schema Registry for Kafka) to enforce and manage schemas centrally, providing a single source of truth.

3. Eventual Consistency

- **Problem:** In an EDA, data consistency is often "eventual." This means that after an event is published, it takes some time for all consumers to process it and update their state. During this window (which can be milliseconds to seconds), different parts of your system might show slightly different views of the data.
- **Mitigation:** Design your user experience and business processes to tolerate eventual consistency. For critical, immediate consistency needs, re-evaluate if an EDA is the right fit or if a smaller, synchronous boundary is needed. Communicate consistency expectations to users where appropriate.

4. Over-Engineering and Premature Optimization

- **Problem:** The allure of EDA can lead developers to apply it everywhere, even for simple, tightly coupled operations that would be better served by a direct API call. This adds unnecessary operational overhead, latency, and complexity without proportional benefits.
- **Mitigation:** Start simple. Evaluate if the benefits of decoupling, scalability, and resilience truly outweigh the added complexity for a given feature. Don't build an event-driven system unless you have clear requirements that justify it. Remember that increased flexibility comes with increased management overhead.

5. Ordering Guarantees

- **Problem:** While some brokers (like Kafka) offer strong ordering guarantees within a single partition, ensuring global ordering across an entire system is extremely challenging and often unnecessary. If events related to the same entity are processed out of order, it can lead to incorrect state.
- **Mitigation:** Understand when strict ordering is truly required (e.g., financial transactions). Design your system to leverage broker features (e.g., using a consistent key for partitioning in Kafka to ensure all events for a given `order_id` go to the same partition). For most scenarios, "causal ordering" (events related to the same entity are processed in order) is sufficient.

Summary: Mastering Reactive Systems

You've now explored the powerful world of Event-Driven Architectures. This paradigm is a cornerstone of modern distributed systems, enabling applications to be:

- **Decoupled:** Services operate independently, reducing ripple effects of change and allowing for autonomous development teams.
- **Resilient:** Failures in one service don't necessarily bring down the whole system; the broker can buffer events, and retries can recover.
- **Scalable:** Individual services can scale independently based on their specific event loads, allowing for efficient resource allocation.
- **Responsive:** Asynchronous processing allows for quick feedback to users while background tasks run, improving user experience.

We've learned about the roles of event producers, brokers, and consumers, and delved into critical concepts like idempotency and Dead-Letter Queues for robust processing. We also examined advanced patterns like Event Sourcing, CQRS, and Sagas, understanding their benefits and their inherent complexities. Finally, we saw how AI agents naturally fit into this reactive paradigm, enhancing system intelligence and automation.

The key takeaway is to embrace the power of events wisely. EDA is a transformative approach, but it introduces its own set of challenges. By understanding the principles, the tools, and the trade-offs, you can design and build systems that are truly reactive, scalable, and robust for years to come.

Next, we'll turn our attention to how we manage and automate the underlying infrastructure that supports these sophisticated architectures, ensuring our event-driven systems can run reliably in production.

References

- Microservices Architecture Style - Azure Architecture Center
- Apache Kafka Documentation (v3.7.0)
- AWS Simple Queue Service (SQS) Documentation
- Azure Service Bus Documentation
- RabbitMQ Concepts
- Pattern: Event Sourcing - microservices.io
- Pattern: Saga - microservices.io

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant. +++

CHAPTER 08

The Sidecar Pattern: Enhancing Services with Auxiliary Processes

Imagine you're building a fleet of microservices, each handling a specific business function. Soon, you realize almost every service needs to do similar things: log its activities, collect performance metrics, handle authentication, or secure its network communication. How do you implement these "cross-cutting concerns" without duplicating code, creating maintenance nightmares, or tightly coupling your services to specific technologies?

This is where the **Sidecar Pattern** comes into play. It's a powerful architectural pattern that helps you enhance your services with auxiliary processes, keeping your core application logic clean and focused. By the end of this chapter, you'll understand what the sidecar pattern is, why it's so valuable in modern distributed systems, and how it can simplify the development and operation of complex applications, including those leveraging AI and agentic workflows.

To get the most out of this chapter, a basic understanding of microservices and containerization (like Docker or Kubernetes) will be helpful. We'll be building on concepts like service-to-service communication from previous discussions.

Understanding the Sidecar: Your Service's Trusty Companion

At its heart, the sidecar pattern is about running a helper process alongside your main application. Think of it like a motorcycle with a sidecar attached: the motorcycle (your main application) handles the primary journey, while the sidecar (the auxiliary process) carries extra gear or provides additional support, sharing the same ride.

What is the Sidecar Pattern?

The sidecar pattern involves deploying an application's components into separate containers (or processes) that run on the same host and share the same lifecycle. While the main application container handles the core business logic, the sidecar container takes on supplementary tasks. This co-location is key: they are deployed together, started and stopped together, and share network and storage resources.

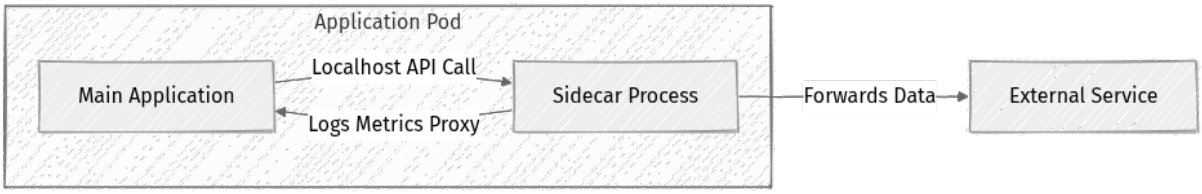
Why does this pattern exist? It addresses the challenge of managing cross-cutting concerns in distributed systems. Without sidecars, you might:

1. **Embed libraries:** Add logging, monitoring, or security libraries directly into each service. This leads to language-specific implementations, increased application bundle size, and potential "dependency hell."
2. **Centralize via a proxy:** Route all traffic through a central proxy, which can become a bottleneck or single point of failure.

The sidecar offers a middle ground, providing localized, dedicated handling of these concerns without burdening the main application.

How Sidecars Work Their Magic

In a containerized environment like Kubernetes (as of 2026-05-15, Kubernetes v1.30), a sidecar is simply another container within the same Pod as your main application container.



Core Mechanics:

- **Shared Environment:** Both containers in a Pod share the same network namespace and can communicate with each other via `localhost`. They can also share storage volumes.
- **Independent Processes:** Despite sharing resources, they run as separate processes, each with its own resource limits (CPU, memory) and distinct responsibilities.
- **Shared Lifecycle:** The Kubernetes orchestrator manages the Pod as a single unit. If the Pod starts, both containers start. If it stops, both stop.

This co-location allows the sidecar to transparently intercept or augment the main application's behavior without requiring direct modifications to the application code itself.

Common Use Cases: Where Sidecars Shine

Sidecars are incredibly versatile. Here are some of the most common and impactful ways they're used in modern architectures:

1. Observability: The Eyes and Ears of Your System

📌 **Key Idea:** Sidecars can standardize how logs, metrics, and traces are collected and sent from your applications.

Instead of each service needing custom code or libraries to push data to a central logging system (like Elasticsearch, Splunk, or cloud-native solutions), a sidecar can handle it.

- **Logging:** The main application writes logs to a shared volume (e.g., a file), and the sidecar container tails that file, processes the logs (e.g., adds metadata, formats), and forwards them to a centralized logging platform.
- **Metrics:** A sidecar can expose an endpoint that collects metrics from the main application or scrape metrics directly from it, then push them to a time-series database like Prometheus or a cloud monitoring service.
- **Tracing:** Sidecars can inject tracing headers into outgoing requests and collect span data, sending it to a distributed tracing system like Jaeger or Zipkin.

2. Configuration Management: Dynamic Settings on Demand

A sidecar can be responsible for fetching and refreshing configuration from a central configuration service (e.g., HashiCorp Consul, AWS AppConfig, or Kubernetes ConfigMaps). When the configuration changes, the sidecar updates a shared file or triggers a hot-reload mechanism in the main application. This ensures services are always running with the latest settings without needing a full restart.

3. Security: Fortifying Your Services

Sidecars can enforce security policies without the main application needing to be aware of the underlying mechanisms.

- **Authentication/Authorization:** A sidecar can intercept incoming requests, validate tokens, and perform authorization checks before forwarding the request to the main application.
- **Mutual TLS (mTLS):** In a service mesh context (which we'll discuss in a later chapter), sidecars like Envoy handle mTLS automatically, encrypting all service-to-service communication transparently.
- **Secrets Management:** A sidecar can retrieve secrets from a secure vault (e.g., HashiCorp Vault, AWS Secrets Manager) and expose them to the main application via a shared volume or environment variables.

4. Networking and Service Mesh: Smart Traffic Management

This is perhaps the most well-known application of the sidecar pattern. Tools like Istio, Linkerd, or Consul Connect deploy a proxy (often Envoy) as a sidecar alongside every service. These sidecar proxies form a "service mesh" that can:

- **Traffic Routing:** Apply rules for routing requests, A/B testing, or canary deployments.
- **Load Balancing:** Distribute requests across multiple instances of a service.
- **Resilience:** Implement retries, timeouts, and circuit breakers for robust service-to-service communication.
- **Service Discovery:** Enable services to find and communicate with each other without hardcoding addresses.

5. AI/Agent Workflows: Streamlining Intelligent Applications

As AI and agentic systems become more prevalent, sidecars can play a crucial role in managing their unique operational challenges:

- **Prompt Management:** A sidecar could manage prompt templates, versioning, and inject dynamic context into prompts before sending them to an LLM API.
- **Token Counting/Rate Limiting:** For cost-sensitive LLM interactions, a sidecar can count tokens, enforce rate limits, and even cache responses to optimize API usage.
- **Observability for Agents:** Sidecars can specifically capture agent thought processes, tool calls, and LLM interactions, forwarding them to specialized observability platforms for AI.
- **Data Pre-processing/Post-processing:** A sidecar could handle vector embedding generation for a knowledge base, or format LLM outputs into a structured format before the main agent processes them.

Step-by-Step Implementation: A Logging Sidecar in Kubernetes

Let's illustrate the sidecar pattern with a common scenario: a simple "Hello World" API service and a dedicated logging sidecar. We'll use Kubernetes to define a Pod with two containers: our main API and a Fluentd logging agent. This example is conceptual; in a real scenario, you'd replace placeholder images and configurations.

First, we start with the basic definition of a Kubernetes Pod. This tells Kubernetes that we want to run a collection of containers together.

```
# Step 1: Define the basic Pod structure
apiVersion: v1
kind: Pod
metadata:
  name: my-api-with-logging-sidecar
spec:
  # ... containers and volumes will go here
```

Explanation:

- `apiVersion: v1`: Specifies the Kubernetes API version we're using.
- `kind: Pod`: Declares that we are defining a Pod, the smallest deployable unit in Kubernetes.
- `metadata.name`: Gives our Pod a unique name for identification.

Next, our main application and the sidecar need a way to share data, specifically log files. Kubernetes `emptyDir` volumes are perfect for this temporary, shared storage within a Pod.

```
# Step 2: Add a shared volume for logs
apiVersion: v1
kind: Pod
metadata:
  name: my-api-with-logging-sidecar
spec:
  volumes:
    - name: logs-volume # A unique name for our shared volume
      emptyDir: {}     # An empty directory created when the Pod starts
  # ... containers will go here
```

Explanation:

- `volumes`: This block defines the volumes available to containers in this Pod.
- `name: logs-volume`: We give our volume a descriptive name.
- `emptyDir: {}`: This creates a temporary, empty directory on the node where the Pod is scheduled. It exists only for the lifetime of the Pod and is excellent for inter-container communication via files.

Now, let's add our main application container. This container will run our "Hello World" API service and will be configured to write its logs into the shared `logs-volume`.

```
# Step 3: Define the main application container
apiVersion: v1
kind: Pod
metadata:
  name: my-api-with-logging-sidecar
spec:
  volumes:
    - name: logs-volume
      emptyDir: {}
  containers:
    - name: my-api # The main application container
      image: your-org/my-api-service:1.0.0 # Replace with your actual
application image
      ports:
        - containerPort: 8080 # The port your API listens on
      volumeMounts:
        - name: logs-volume  # Mounts the shared volume
          mountPath: /var/log/my-api # Path inside the container where the
volume is mounted
      env:
        - name: LOG_PATH
          value: /var/log/my-api/app.log # Environment variable for log file
path
      command: ["/app/my-api", "--log-file", "$(LOG_PATH)"] # Example command
to start app
```

Explanation:

- `containers`: This list defines the containers that will run within our Pod.
- `name: my-api`: Our main application container.
- `image`: The Docker image for your API service.
- `ports`: Exposes the application's port.
- `volumeMounts`: This is crucial. It connects our `logs-volume` to a specific path (`/var/log/my-api`) inside the `my-api` container. The application will write its log files here.
- `env`: We define an environment variable `LOG_PATH` to tell our application where to write its logs.
- `command`: An example command that starts the `my-api` application, instructing it to use the specified `LOG_PATH`.

Finally, we introduce the sidecar container. This container will run a logging agent (like Fluentd), which will read the logs written by our main application from the shared volume and forward them to a central logging system.

```
# Step 4: Add the logging agent sidecar container
apiVersion: v1
kind: Pod
metadata:
  name: my-api-with-logging-sidecar
```

```
spec:
  volumes:
    - name: logs-volume
      emptyDir: {}
  containers:
    - name: my-api
      image: your-org/my-api-service:1.0.0
      ports:
        - containerPort: 8080
      volumeMounts:
        - name: logs-volume
          mountPath: /var/log/my-api
      env:
        - name: LOG_PATH
          value: /var/log/my-api/app.log
      command: ["/app/my-api", "--log-file", "$(LOG_PATH)"]
    - name: logging-agent # The sidecar container
      image: fluent/fluentd:v1.16-debian-1.0 # Using Fluentd as our logging
agent (checked 2026-05-15)
      volumeMounts:
        - name: logs-volume # Mount the *same* shared volume
          mountPath: /var/log/my-api # At the *same* path as the main app's
logs
      # Example command for Fluentd to tail the log file and forward it
      command: ["fluentd", "-c", "/fluentd/etc/fluent.conf"]

# A simplified fluent.conf (not part of the YAML, but conceptual for the
sidecar):
      # <source>
      #   @type tail
      #   path /var/log/my-api/app.log
      #   pos_file /var/log/my-api/app.log.pos
      #   tag my-api-logs
      #   <parse>
      #     @type json # Or other format like regexp
      #   </parse>
      # </source>
      # <match my-api-logs>
      #   @type stdout # For demo, output to stdout, normally to a remote
service like Elasticsearch
      # </match>
```

Explanation:

- `name: logging-agent`: This is our sidecar container.
- `image: fluent/fluentd:v1.16-debian-1.0`: We use a specific version of Fluentd, a robust open-source data collector, as our logging agent.
- `volumeMounts`: Crucially, it mounts the same `logs-volume` at the same path (`/var/log/my-api`). This grants the sidecar access to the log files written by `my-api`.

- `command`: Fluentd is started with a configuration file (`/fluentd/etc/fluent.conf`). This configuration (conceptually shown in the comments) tells Fluentd to:
    - `tail` (continuously read) the `app.log` file.
    - Parse the log entries (e.g., as JSON).
    - Forward them to a specified destination (e.g., a central logging service like Elasticsearch, or `stdout` for demonstration).

This setup ensures that `my-api` focuses solely on its business logic, while `logging-agent` handles the complex task of reliable log collection and forwarding, completely decoupled from the main application's codebase. The sidecar standardizes log handling across potentially many different services, regardless of their implementation language.

Benefits and Tradeoffs: Is a Sidecar Always the Answer?

Like any architectural pattern, sidecars come with their own set of advantages and disadvantages. It's vital to understand these tradeoffs to apply the pattern judiciously.

Advantages:

- **Decoupling:** The main application remains focused on business logic, free from cross-cutting concerns like logging, monitoring, or security.
- **Reusability:** Sidecars can be developed once and reused across many different services, even those written in different programming languages. This promotes consistency.
- **Independent Development:** Sidecars can be developed, deployed, and scaled independently (within the Pod) from the main application. This means you can update your logging agent without touching your core service code.
- **Polyglot Support:** A sidecar can be written in a language best suited for its task, regardless of the main application's language. For example, your main service might be in Java, but its sidecar can be a Go-based proxy.
- **Reduced Cognitive Load:** Developers of the main service don't need to worry about the intricacies of logging, tracing, or security protocols; the sidecar handles it.

Disadvantages and Tradeoffs:

- **Increased Resource Consumption:** Each sidecar is a separate process, consuming CPU, memory, and network resources. This can significantly increase the overhead per Pod, potentially impacting the cost and density of your deployments.
- **Operational Complexity:** While simplifying application development, sidecars add operational complexity. You now have multiple containers to monitor, debug, and manage within a single Pod.
- **Shared Failure Domain:** If the sidecar fails catastrophically, it can impact the main application, as they share the same Pod lifecycle. For example, a misconfigured sidecar crashing repeatedly might cause the entire Pod to restart.
- **Network Latency (Minor):** While `localhost` communication is very fast (typically microseconds), it's not zero-cost. For extremely high-throughput or ultra-low-latency applications (e.g., certain financial trading systems), this small overhead might be a consideration.
- **Over-engineering Risk:** For simple applications or those with very few cross-cutting concerns, a sidecar might introduce unnecessary complexity and overhead. Always evaluate if the problem you're solving truly warrants a dedicated sidecar.

Mini-Challenge: Configuration Sidecar

Challenge: You are developing a microservice that generates daily reports. This service needs access to a dynamic list of email addresses to which these reports should be sent. This email list changes periodically and is managed by a central configuration service (e.g., a simple HTTP endpoint that returns JSON).

Design how you would use the sidecar pattern to provide this dynamic email list to your main report generation service. Sketch out the interaction and the role of each container.

Hint: Consider how the sidecar would fetch the list (e.g., poll the HTTP endpoint) and how it would make that list accessible to the main application. Think about shared file systems for configuration files, or perhaps a very lightweight local HTTP endpoint exposed by the sidecar itself.

What to Observe/Learn: This challenge helps you internalize how sidecars can abstract away external dependencies and provide dynamic updates, allowing the main application to remain simple and focused on its core task. You'll also think about communication patterns between the sidecar and the main application.

Common Pitfalls & Troubleshooting

Even with the best intentions, sidecars can introduce new challenges. Here are some common pitfalls and tips for avoiding them:

- **Over-engineering for Simple Needs:**
    - **Pitfall:** Deploying a sidecar for every minor cross-cutting concern, even when a simple library or direct integration would suffice.
    - **Troubleshooting:** Before adding a sidecar, ask: Is this concern truly complex enough to warrant a separate process? Does it need polyglot support? Will it be reused across many services? If not, a simpler approach might be better. For example, if your application only needs to print logs to `stdout` and your container orchestrator handles forwarding, a logging sidecar might be overkill.
- **Resource Bloat:**
    - **Pitfall:** Neglecting to monitor the resource usage (CPU, memory) of your sidecars, leading to increased infrastructure costs and reduced cluster efficiency.
    - **Troubleshooting:** Implement robust monitoring for all containers within your Pods. Use Kubernetes resource requests and limits to constrain sidecar resource usage. Optimize sidecar images (e.g., use smaller base images) and configurations to be as lightweight as possible.

- **Debugging Distributed Failures:**
    - **Pitfall:** When an issue arises, you now have two (or more) processes to inspect within a single Pod, making root cause analysis more complex.
    - **Troubleshooting:** Ensure you have centralized logging, metrics, and distributed tracing enabled for both the main application and its sidecars. When troubleshooting, look at the logs and metrics of all containers within the affected Pod to understand their interactions and identify the point of failure. Tools like `kubectl logs <pod-name> -c <container-name>` are essential.
- **Misunderstanding Shared Lifecycle:**
    - **Pitfall:** Forgetting that if a sidecar crashes repeatedly, it can cause the entire Pod to restart, impacting your main application's availability, even if the main application itself is stable.
    - **Troubleshooting:** Design sidecars to be robust and handle their own failures gracefully. Implement proper liveness and readiness probes for sidecars in Kubernetes to ensure they are healthy and ready to serve before traffic is routed. If a sidecar is non-critical, consider configuring its restart policy carefully.

Summary: The Power of Modular Enhancement

The sidecar pattern is a powerful tool for building resilient, maintainable, and scalable distributed systems. It enables you to:

- **Decouple cross-cutting concerns** from your core application logic, promoting cleaner code and easier maintenance.
- **Standardize operational practices** like logging, monitoring, security, and configuration across diverse services, regardless of their implementation language.
- **Leverage specialized tools** (like Fluentd for logging or Envoy for networking) without tightly coupling them to your application's specific language or framework.
- **Enhance AI/agentic systems** by offloading tasks like prompt management, token counting, or specialized observability, allowing the core agent to focus on intelligence.

However, it's not a silver bullet. Always weigh the benefits of modularity and reusability against the increased resource consumption and operational complexity. The key is to apply the pattern judiciously, understanding when its advantages truly outweigh its costs for your specific problem.

What's Next? Many of the advanced capabilities of sidecars, particularly in networking and resilience, are foundational to understanding **Service Meshes**. In our next chapter, we'll dive into what a service mesh is and how it leverages the sidecar pattern to provide powerful traffic management, security, and observability features across an entire fleet of microservices.

References

- Microsoft Azure Architecture Center. "Microservices Architecture Style." Microsoft Learn, learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices. Accessed 15 May 2026.
- Kubernetes Documentation. "Pods." Kubernetes.io, kubernetes.io/docs/concepts/workloads/pods/. Accessed 15 May 2026.
- Fowler, Martin. "Sidecar." MartinFowler.com, martinfowler.com/articles/microservice-patterns/Sidecar.html. Accessed 15 May 2026.
- Fluentd Documentation. "What is Fluentd?" Fluentd.org, fluentd.org/architecture. Accessed 15 May 2026.
- Istio Documentation. "What is Istio?" Istio.io, istio.io/latest/docs/concepts/what-is-istio/. Accessed 15 May 2026.

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 09

Observability: Logging, Metrics, and Distributed Tracing

Imagine your beautifully crafted distributed system running in production. It's composed of many microservices, perhaps handling millions of requests per day, or coordinating a fleet of AI agents. Suddenly, a customer reports an error, or a critical business process slows to a crawl. How do you find out what's going on? Where do you even begin looking?

This is where **observability** comes in. It's the ability to infer the internal state of a system by examining its external outputs. In complex, distributed systems, you can't just attach a debugger to a single process. You need to gather data from every corner of your architecture to piece together the full story. This chapter will equip you with the fundamental tools and mindset for achieving deep visibility into your systems: logging, metrics, and distributed tracing.

We'll explore what each of these pillars entails, why they are indispensable for modern architectures—especially those involving dynamic AI/agent workflows—and how they work together to provide a holistic view. By the end, you'll understand how to design systems that are not just functional, but also transparent and debuggable.

The Pillars of Observability: Logs, Metrics, and Traces

In the world of distributed systems, "monitoring" often means checking if a service is up or if a CPU is spiking. **Observability**, however, goes deeper. It's about being able to ask any question about your system's behavior without deploying new code. It's about understanding why something happened, not just that it happened.

For AI-powered systems, observability is even more critical. AI agents often make complex, multi-step decisions, interact with various tools and services, and operate autonomously. Understanding their internal reasoning, tool usage, and performance bottlenecks is paramount for reliability and trust.

The three pillars of observability—logs, metrics, and traces—provide distinct yet complementary insights into your system's health and behavior.

Logging: The Storyteller of Events

What is it? Logs are timestamped records of discrete events that occur within your application or infrastructure. Think of them as the narrative of your system, detailing specific actions, decisions, and outcomes.

Why is it important? Logs are invaluable for debugging specific issues, auditing system behavior, and understanding the exact sequence of events that led to a particular state or error. When a problem arises, logs are often the first place engineers look to find clues.

How does it work? Modern logging emphasizes **structured logging**. Instead of plain text messages, logs are emitted as structured data, typically JSON. This allows for easier parsing, querying, and analysis by automated tools. Key information like timestamps, service names, request IDs, user IDs, and specific event details are included as fields.

For example, a traditional log might look like this: `2026-05-15 10:30:05 INFO User 123 requested product 456.`

A structured log for the same event would be:```json { "timestamp": "2026-05-15T10:30:05.123Z", "level": "INFO", "service": "product-catalog-service", "message": "Product requested", "user_id": "123", "product_id": "456", "request_id": "abc-123-xyz" }

```
This structured format makes it trivial to query for all logs related to `user_id: "123"` or `service: "product-catalog-service"`.

**⚡ Real-world insight:** Logs are typically collected by agents (like Fluentd, Logstash, or Vector) and sent to a centralized logging system such as Elasticsearch (part of the ELK stack), Grafana Loki, or cloud-native solutions like AWS CloudWatch Logs or Google Cloud Logging. These systems allow for powerful search, filtering, and visualization of log data.

**AI/Agent Context:** For AI agents, logs are critical for understanding their "thought process."
- **Agent Decisions:** Log when an agent receives a prompt, decides on a tool to use, or chooses a next action.
- **Tool Invocations:** Record which tools are called, with what parameters, and their raw outputs.
- **Reasoning Steps:** If an agent has a multi-step reasoning chain (e.g., "Plan and Execute"), log each step and its intermediate thoughts.
- **External API Calls:** Log requests and responses to external services.
This level of detail helps debug unexpected agent behavior, evaluate prompt effectiveness, and audit compliance.

### Metrics: The Numerical Pulse of Your System

**What are they?**
Metrics are numerical measurements of data points collected over time. They represent an aggregation of events or a state at a particular moment, providing a quantitative view of your system's performance and health.
```

```
**Why are they important?**
Metrics are ideal for monitoring trends, generating alerts, identifying
performance bottlenecks, and capacity planning. They answer questions like
"How many requests per second are we handling?" or "What is the average
latency of our database queries?"

**How do they work?**
Metrics come in various types:
- **Counters:** Increment-only values that represent a cumulative count (e.g.,
total requests, total errors).
- **Gauges:** A single numerical value that can go up or down (e.g., current
CPU utilization, number of active users).
- **Histograms:** Sample observations and count them in configurable buckets,
allowing for calculation of percentiles (e.g., request durations: 90th
percentile latency, 99th percentile latency).
- **Summaries:** Similar to histograms but calculate configurable quantiles on
the client side.

Applications expose metrics through an endpoint (e.g., `/metrics`), and a
monitoring system (like Prometheus) periodically "scrapes" these endpoints to
collect the data. This data is then stored in a time-series database.

**⚡ Real-world insight:** Prometheus is a popular open-source monitoring
system that collects and stores metrics. Grafana is commonly used to visualize
these metrics on dashboards, allowing engineers to see performance trends, set
up alerts, and identify anomalies. Cloud providers offer similar managed
services like AWS CloudWatch Metrics and Azure Monitor.

**AI/Agent Context:** Metrics are crucial for understanding the operational
efficiency and reliability of AI agents:
- **Agent Latency:** Time taken for an agent to respond or complete a task
(e.g., average, 90th percentile).
- **Token Usage:** Number of input/output tokens consumed by language models.
- **Tool Success/Failure Rates:** Percentage of successful tool calls.
- **Task Completion Rate:** How many tasks an agent successfully completes per
unit of time.
- **Cost Metrics:** Estimated API costs incurred by agent interactions.
These metrics help optimize resource usage, manage costs, and ensure agents
are performing within expected parameters.

### Distributed Tracing: The Journey Map of a Request

**What is it?**
Distributed tracing provides an end-to-end view of a single request or
transaction as it propagates through multiple services in a distributed
system. It visualizes the path, timing, and dependencies of each operation
involved.

**Why is it important?**
Traces are indispensable for understanding performance bottlenecks in
microservices architectures. They help pinpoint exactly which service or
operation is contributing most to latency, identify failing services, and
visualize complex service dependencies. Without tracing, debugging issues
across multiple service boundaries can feel like searching for a needle in a
haystack.

**How does it work?**
A trace is essentially a collection of **spans**.
- A **trace ID** is a unique identifier that links all operations related to a
single request.
- A **span** represents a single operation within that request (e.g., an
incoming HTTP request to a service, a database query, an outbound call to
```

```
another service).
- Each span has a unique **span ID** and references its **parent span ID**,
forming a hierarchical structure.

When a request enters your system, a new trace ID is generated. As the request
travels from service to service, this trace ID (and the current span ID, which
becomes the parent ID for the next operation) is propagated, typically through
HTTP headers. Each service, upon receiving the request, creates new child
spans for its operations and sends them to a tracing backend.

Here's a simplified illustration of how a trace connects different parts of a
system:

<div class="diagram-wrap"><img src="
JREFUeJzsnXl4FMXWxt9TPTMJgQAzkbAjiLKIuCCLC0IQZREERURlkT2sckVxQ/
3wXkRF3BWUBBRFNgWUVVAUF0AERUFQQLmyGQkwMyH7LF3n+2NmYtJTQSJ4Q0j9nofHS1VXdXUnlzp9
6pz3ABqNRqPRaDQajUaj0Go1Wg0Go1Go9FoNBqNRqPRlEk8Hk+Hk+tkzXwEgi4
3DXF9ZX0Wf8K1bOWBbKzs6tnZmY2OtXrmfk9t9v9icfj6f5PrkujKStQaS9AoyktZs6c+TkzXwEgi4
iyAGQxc5bT6bwoPj6+bl5e3rfHjh3bSESZAAKGYXC9evWSDh06tDYQCOQwcwBAAECgRYsW4x0OR/
yhQ4deOHLkyH7TNH2JiYnOZs2azRBCVAIg3G734z/88MN6IpKJiYnxl19+
+RIhRCIRZefm5j6xcePGxcxs3nDDDWOFEFVtNtul6enpo7Zu3bo/
GAya3bp1uzQ+Pv5TIoolon85nc5XIs+SkZHRyTRN4XK5NoafpQC32z2HiAYS0Xin0/
lS4T6Px7MAwJ3MPCwhIWF24b4jR44MFUL0+e9//3v3rl27coPBoDBNU0gpxQ033HCp3W5vsm7dug+l
lEJKKWw2G1177bU3x8fHP52fn5+ya9euuceOHfMxs2Bm0b59+wdN0/
Ru2LBhjs1mE8wszjvvvKoNGjTol5aWtjYtLe2waZrEzIKIDACicePG1/t8vuP79+/
fi9AHkBH5b8OGDZMyMjJ+c7vdRyLtRGQws4iJiYn3+Xz51jEARK1ata7OzMw8mJOTc5yZHURks9lsc
bVr176DiCqlp6evycvLy2JmGwD7eeed1yoQCORlZmYeZGYbERlxcXENY2Ji6jAzpJQIBoN5eXl5R5g
5npkrElEFxa9cFoChI0aMeP/v/
s5qNGcz2qDQlFtSUlJaMfNLAGoCiAdwXikvSXPus0FKeeeoUaN+L+2FaDRnGltpL0CjKS2klJ2J6Bp
mfo6ZVzVp0uTKmjVrPme9bteuXXc2bNiwf2xsbPdCY727d+8ebxhGXMOGDWdIKVHoazUrEAik2+32C
wu3R/6bl5e3OzY2tgkzI/LH5/P95vV6f0hMTLy1cDszAwACgUC2zWarJKUEADAzAoFAttfr/
cbpdF4jhKgQud40zWyv1/uDYRjxVapUuazws5immev1erfFxMQ44+PjmxXuY2bT6/
VuAWA4nc4WRFTw70MwGMzxer3fCiHsVatWbWEYRmykLy8vb39eXt4fVatWvVII4bDeLy8v73ClSpWK
HCX4/f5jgUDgRMWKFS8kIhCFvm2ysrK25eTk/G632x2JiYlJQogYIQSICHl5eT+lp6d/
7XK56rtcro6RcaZpZhw+fHie3+/PuvDCC/vb7fZEIYSDiJCbm/
vtvn37FjkcDtG4ceN77XZ7zch8AOB2u9+Ij49vHhcXdy0ARPpM09yXkZHxfmJi4sNEBMMwIIQAM/
9ss9nq2Wy2ioXbbTYbmDmHiF4B8IjlvS4hoqHvvffeEiLqKIS4FcBrf/HrqdGUObRBoSm3ENGF4f/
uv+uuu343TfND1XXVqlW7jYiKnJMzc6B+/frr8vLyPolsToWID/+Jgpk/
IKJbFe15ADop5opQSdHmZ+ZmCvd6JQA1AKhiGfKY+QEAn1rvJYToarPZfvT7/
Ztg+beBiKZKKV8moi8AxBbuY2Z7eA1FjIkwcQAaWa7PJ6LHAbxhvZiZTZfL1dfr9W4EEGPpvhiAD8A
VlvaqV1555XUAfgZQx9LXskWLFguJ6AKEPFGFODagQYMqAK5VrLth3bp1H1a0N1W0AQCEEI9JKadY3
ysR3cbMzQzD+G/YIOTi5tBoyjI6KFNTnvEBQIUKFRJM01wDICqwkpnfJ6Lbre2GYYzJz89/
jIiK3WAUrADQWdF+gIhqEVHFU52ImbMBeIiohqIvn5lrQXGEw8xPE9EiIoqztCcD+Nbv968D0MAy7F
Wn0zmZiJYBuNw6JxHVVrUXhxBiFIAJiq5DsbGxt3i93kUALi1muNWYADPnMfMxAHcort8ohHADGGEZ
kwPgfQB3neq6AWQU18HMycx8j/
W9RiCiJjVq1GgbvlYbFJpzEm1QaMotzOwDgAYNGtwD4ALFJQtUxgSAVVJKB4CRJbjXWmZuqdjIjyD0
FX/KmRHhL/
wfAVxYTN+RYja2ZxDaQK1ZJc+5XK75pml+DKCZpW+ey+Ua53a7FwNIOtU1FgcRjWfmW6xrZ+ZMZu7i
8/kmAriphNOuIaKO1kZm/omZJ0gpo44XiOh1AKMVY3LDxpqV/
Sj+ZzQaIWNG9TtUQGxsbGUAcDqd1U52nUZTVtEGhabc4nA4JAAIIVTBmIuYuae1MbzxvQNg3qneh5k
/
AeAkopqW9hyE3N8l2mCIaDmAq4vp+xpAfUXXu8zciIiutLQvc7lcD3g8nmVE1NLSt9LlcvX3eDwziO
i2kqyxGF5j5lgAUe/VMIzeQoiOAMaUcM55qiMkAPsdDkdvIpqn8Pw8w8xKYzBsjBU5XmLmPVC/
UxDRJGZuojJorBiGAQCoVKnSSQ0Pjaasog0KTbmlXr16fQAgEugYgZk/AHCj6itfCPE4EUWd/
RcHM39HRF4iam1pzweQYTUy/goimgWgTzHdCwF0ULR/SUTpRNTL0r7N6XT2d7vdi60bIjN/
63Q6+3g8nokARpVkjcWwTAixHMDT1g4iGiKltDPzK4pxJ2MRM9+saD9us9m6+P3+V2HxGhDR88w80G
o0hFlvvZ6Z9xBRYjH3T2HmdCIadyqLtdvtAIDDhw9vOZXrNZqyhjYoNOWW3NzcbwEUZFKE//
cHAK4C4LJez8wvSinHAnCe4i1+JaJPoDYAjodjD6IIey5U7bOllKo4ATDzm8VsrvsBvMfM91uuP+xw
OLp5PJ5XFd6HA3a7vavX670NwBTV/UrItri4uPuklAsVfU8x83fM/
F4J51zGzG2JqHLhRmbOJaIbA4HAZIWRNIeZO6uMOGZ+ExZjjJn/IKIYqH/
```

```
ea4hoIRSBpcUhpTx0qtdqNGURbVBoyi1paWlfAn96KIjoeQCNi9lw1hJRSyK66BSnP0ZE0wFEZQow8
2EismYjRPr+KCY4cwERdSSiqOwRZp4D4DrruPDxzBRYUhSZOcfhcHTy+/
3jiWiQZTqvYRg3BAKBlgDmnOT5TpWDsbGxPXJzc5fAYqQx8wexsbEzmPnjkgSkAvgCwPkqg0wI0Y2Z
h1ljX5h5MxE1AHCJdUzY69Pbcn0mEf0B9VHH9zabbbyUUpkVpIKZ/9i/f/
+rp3q9RlMW0QaFpjzjA4ATJ05sAtCNmVsQ0cXWi5h5L0JplNedyqTMnM3ME6WUUe79vzAmfi7mCGQl
M18E9ea2EEAtlaFDRBMBvGBtF0L0DwaDnQE8aLl/nmEYXRGK9/gIIWXJ0+G4YRgd8/
PzX0B0FsgGm802LD8/fzURVT/VCZn5m/
BxUVRWCTP3kVK2QHQcxn4icgNor5hynpSyndXTQUQbALRQ3OMXIcTtgUBguXXMyTAMowcRRbJEtKCg
5pxEGxSacgsR+QDgyJEjW5h5EBTxB2GPwZFTCbqLIIT4F4DJRBRr6TpenDEB4IdiUlA3hddqDZgEgD
XMfISIOin6ngYwXuHReIyZ45j5ResAIroDgMc0zY+KWWOJMAzjJinlXbAc+TDzL3a7vVcwGFyA4tND
owgHR/
5ERKrU21FEZA97mQqPyQewA0A3xXxLAdQmImv9jtVQZ5p4bDbbTVLK1BJ4qiIxIrc1b958Qvi+p2uo
aTRnJdqg0JRbwpsNatSo0V6VHsrMaUS0G8ApF+Ni5uHMPN6qD8HM6cys1CgAsJOZVSmgPwPYDiCq+B
Qzbw0rMN6rmG8+gOYAGlrbhRBboM5QGelwOLaYprkWQEIx6zxlmPlO0zQbM/N/
LF1uwzC6+P3+Z4oxhIqb7wgRLSWiwYq+yUKIX6F4LiJaB6CHYsxnCOlKJFm6NkKRQcPMuUKITqZp3g
914Gtx636RQ/
ViHo6JiWkEABdffPE9zKwUPtNoyjLaoNCUZ3wAEBcXpxJLOgJgN0qweQB4CKGv8SLn9Mx8lIiMYrQh
fmVmpyJVMR3Am1BnWPwXwONElKro+5KZj8NihDDzZmZ+rRg10KecTuc8v9+/
BtGiViWGmf8thPgDwFxLez4z3yylvIOIhpRgviwiegEWSetw3xwhxFLTNJcp+tZBYYwh5A361LqGsG
5FIhRBmETUh5mvRQm0RxDKGtkF4B7gzywPu93e0Ov1zijBPBpNmUAbFJpyS/369RsBRbM8wn8/
QkRbiej6U50rnPJ4IRHdaOk6QUTZUBceO4hQpdMiwYXMnC2EeJqIpinucwTA3UQ0V9H3M4CV1jRGZj
4M4D4iWqYwat51uVyPejyepSiB2mVxMPP7DodjETMvt/YZhtE/HBj5VEnmJKKJzPx/
iq5VhmE8GQ7qtAqGbQagKtl+iIhegiV7JRzbcqKYo4yRAIiZXy7Bsg8w89RwwCeAP3UoAoFAZmxsrE
rWW6Mp02iDQlMuyczMbFSvXj2rOx4AdhLRFgBRKZjhzVzFQiJKBzBc0XcAagXFDAA/QSElTUSPMPO/
FffPNgzjZgDTES2G5QbwIoBnLWPyDcMYTETvWscw86cul2uAx+OZrzCE/
g7bhBAPBgKBj2CRMWfmB8Kek1MWBAtzD4AJCg/
OVmYeJaVcg+jn+pmIGimMpxNENE5K+arl+lwAX0MtFjaViDYzsyrlVUnYEzMCliwZmy1UHuXgwYNvx
8XF6WqjmnMObVBoyiWVK1feGwgE1gFFhK1WMvMvUJ+5b7bGRYTbPwlnRKj0Gjai+KDDVQC6KNofBHA
/FHVFhBA3m6b5CIAiFUTDG+K/
AERVSgUwSEr5IhSCTS6X61aPxzMNJatnoYSZDzudzt5hz8T5lu4Uu92+Qkq5ooTTPsbMAxXz7SOi3h
SqLWKV8E4jovOg1hG5Q0r5qiJQdaYqhgbA8piYmJellGtKmNZ6NxE9bf19iXgosrOzvSWYS6MpM2iD
QlNu2blz51NAwZHHcwAkqWWclxPRVYr2bUT0MjO/
rej7EuoqlgCQCqCfov1ZALdCkR4a3gyvV6hdRoptTVGkMU4EMAzRMR1HANyQkZExGOoiXSUifETT3e
v1TkcoGLQwK2NiYp4IBoNrVRoaJ2EGM1+myG45ZhhGV2ZOgcW7Ez6qCkAhZc7MdxDR/
ymybFIAJCvuv1EIMcjn861WGZLFwcxPIpRREuV5CgQC/
QDAbrdbq8NqNOcE2qDQlFuys7NPAEBGRsYWAK2g8EwAeA/qs/
hfKVToar61I+yOV8YjhI0PVV2MeQh9iUe53SOZHBQq+W1ljJRyHCxf8cz8JDNfQUQ3WNpzw2mX15Qw
JqBYDMPoJaW8B0BXS9c2u90+xOfzrbKu72Qw82IAXoU4VS4RdTJNc5IidTQDgKeY+4wmop6w/
BzDaaMdFYJge202W08p5TsoWVzJSgpJnA9U9G2KjY2tCAD169cf5PV665dgXo2mTKANCk15JgAAcXF
xjaAWPVoEoDEsAZXhFNABUsr5Cq/AjwgJTalEj5YDaI1od/
x6AIehKL9NRM8T0VdQqFYS0RMAulB0wa+FQog4lRtfCNGNiFxEtEixvhJDREOllFcS0dDC7cx8ODY2
toff758Lxdf6SdgIYA2AR60dhmHczMx3QuHdYeZfVKJkCFVYdQHoa2n/
nEI1OqyptW7DMLoEg8HJUGeIKOFQzY9UACo1zK8AXONwOFKEECCiasy8zev1Jp3q/
BpNWUAbFJpyS6NGjRoCgGEYqrLUkSyFqHgFIrqNiN5QSD/
vZ2ZD0Q4Am5i5glW8KhxA+AFCKadWFkkpXzJNcxURWd3kqeEUxyLBo+Hshu3MfJ9ivgFSymNSyqgUy
7/
Js8ycB0vBL2bOJqKb8vPzVZ6EkwW37gLwTOHMiEL0k1JeDMV7YuZPiaiVYsy7QojvADxpuX4PgHQAb
S3teeHjlJ4oWUE0r2EY93BIAt3KFwAKDB3DMCJHbE5mLk7kTKMpk2iDQlMuyczMbFK/
fv15QHTaKICvmPlzKDwGAHox8xOwGBoIudsPFPOV/CuAHxWZFMeI6ClWV9lcz8zJRLRKcYa/
Mrwpjra0HyCid6Co6AngsdjY2PUA1hXjPSkRzLyUiFYhJKJVBCLqg9DXvSrr5ftiglt/
Ryg9MyoehZknMHMA6q//VaRQMWXmdcw83TRNa3qtByElzKifrRCiv5SyFitURP+CgaZpvoZo/
YqvmLkGCgmF2Ww2MDOY+T8ul+vdEt5Hozmr0QaFplxSuXLl3VLKH4CoaqPfCCGeDAspWelLRP2tcQn
hcV9AfWySEdaMGGG5Pn/
37t2PuFyuuTNmRGkc7UxISGjQp0+fXbBkiTDztwDmE9FzlvYshAwJlWDSG06n85W8vLyPTyXAcOTIk
XC5XHC5XKhXrx6SkpLw3HPPIT8/
v2ANQojJzKwSyRqF0MYapTXBzJ9wqCaJtT0bwO3MPBvRx0HThRDfEZGqGukqKCS1EVIXvZeIllNY/
lxKiV9+
+QW5ubkziWi8Yg33E9FBKeUCxXzFQkRDmHkMRct3/8DMfiJqXLjRMAzk5ub+mpCQMKkk99FoygLaoN
CUW7Zv3279wt9mt9sHmqb5juLyh5i5KYD+1g5mnlNMdkgkODJKU0IIMaJ69epRQZbM/
LvD4bixatWqzoSEhDqWvsNE9H/
hjdc63+MApirm+8DpdN7n9XpXFeM9UVKrVi0sX74cqamp6NChA2bNmoW+ffvCNM1DAAZyqMx7kS9yI
nqBQlLlKq2JDwFcbNWTCK/9ZgBPKjblD4koVXVEw8xroQiWZebDhmHcDuB9FMr2mD9/
```

```
Ptq0aYPMzMyxirW9LoRYKqVcrThaOhnPMnNTxbHOfgDfqDwnALLT0tI2lOAeGk2ZwVbaC9BoSgu325
0JAIFA4AQz/
2y3228OBoMfUnT1y9eY2U1Eqg17NhGpsjYiQZNRwYXh7JARQghrRsIJh8NxYyAQ6FS5cmWrDsUJIho
J4B3rpsfMk6HWrtjgcrnu9Hq970NRKTWsXyFVm3yFChXQtm0oxKBz585o2LAhxo0bh7Fjxz75+uuvz
4UltZWZP7DZbKmBQGALUdFimuHjkUaq2BIi6imlvJuiVUm/
F0Lcb5rmJusRTVj7oymijxgyAHQOBoPTrbEqBw4cWA+ggzV1lZk/
EUJMZOatUKSbnoSVRLSTma3GpxvATCiOnZh5T05OjpeI9L+7mnMS7aHQlFuklCYAHD9+fANCG9HLiN
aOWAbgY1WgIIeKc7UEoArqnBEWZbJ+xT/PzK2h/
rq+KRgMVlPpWixZsuSRHj16vNesWbPzGjVqhF69emH16tUgollEdOuxY8fqjhs3Ds2aNUPz5s0xZsw
Yz5YtW3p6PJ5nAfTYvHkzXC4Xfv31V4wdOxZ16tTBzp07c1TGhIqmTUP780UXXTQa4bLeW7duxYABA
3D++ecHa9So0Xz69OmbrBv2q6++
+k379u07NGjQ4JJBgwZh1apVcLlcyM7OBoAxBw4cuDQhIWHw22//+cjffffdQZfLdcWHH374RcS4M0
0TU6ZMQZs2bbLOP//86++66656x44dKxjj8/
lw6aWXft20adOv69evQkqJ5s2bZz3//PMdIs/
hcrkQCAQA4EfDMO4IH91EFWc7CduJKFVhTADAo6yQCWfmwxUqVOgYPt6JKcG9NJoygzYoNOWWiEHBz
ATgPlhSC5n5OwDPs0J2OZxNIREdnAmE3PttYSm0xcxLpJTZUChTCiF6G4bxR/gowcrIt95666njx4/
HPfLII5g2bRpq1KiBpUuXHmTmZAuGTRoEFasWIFhw4Zh1KhRGYsWLTLTJmylIj+VXii5ORkVKhQA
U8//fThpk2bnvIX+fz582cCQLNmzS4DgOPHj6Nv377UuYxNnoEaNGo8MHjw4Zh
vHHH3euWrWqYMxHH330xaRJk9qcf/75ztdeew1XXHEFHn005LDJz89/
CcChKlWqTLbc6viiRYvGMeOHcPtt99eeP/St2/
fuenp6R3HjRtXOSUlBVdffTXrFmDQCDw9bPPPvvVv/
4Veg3vvvPMOVq9eDZvVdoSZu4VlufWFxWL27VewhnqEyUUUqoqmw5l5scUxyZeAJ3z8vKqxsfH12Rm7aHQ
nJPoX2xNuUVKaQohUKlSpRpEZA2S22+2z2Uabprmaomt6C/
CaE+EgVGwHga2auSERWye2vhRCrmPlN64AffvhmZRyHTNvIaIiQYlh5cUb9+3bV7VLy7o3z8UwtG
zZ88dCBUX675p0yZ8/
fXXeOmllzBgwIBMIrrut99oaL7MjMzUbnynycGtWrVwrRp075ECUqy79mz58WPNPPvqod+3atZG
UlAQAePvtt+F2u9F+Z24AsHrq1Kl16tWrzhL+cmAz6hKBmUmmU6CHzGdtheubwmzvWVrRj1rNnzJcfhnVrd3f9uthUDaU...
```

```
vcunXrF+edd951AGIBoEmTJujXrx9ee+01MykpSV533XWPDRkyBKmpqRg2bBjGjBmDFStWTDtw4MCK
li1buh544AEAwODBg3HLLbfg0ksv/WbJBUDUAAAgAElEQVTYsGH/IqLBhYMtAWyKj4+fd/
HFF9+6detWDBkyBLt27cKkSaH42FatWg0AkBAXF4d77rkH06ZNQ2JiYs6aNWv+fckll3zy6aefNlu8
eDFcLhfuuOMOMPMv1113Xecrrrii1axZs1C1alW0atUKzLz3448/HgPgs/fff7/
uZZddhqZNm6JmzZCjiYhehPrn/
huAjUT0gOVnnmWz2e40TXM+RVcqfctutz8XDAY3U6j4WBHS0tLmh9OGNZpzDu2h0JRbFi5c+DoA+P3
+Y8w8HYpNhZl/
YubJzJyqmGIDgLYK4aXZAGojunBXnmEYNxmG4YjIVvfu3RszZ85Eeno6hg4d6lu1alX/
1q1biwULFsButwMAEhISdr/+
+us5J06ciJkwYQIeeeQRJCUl4d133/3G5XINYubF8+bNq9ymTRtMmDABAwcOxK5du9CrV69fEDrft2
prFIsQonfjxo2bxcTExBZuv//++79i5uArr7zyGADEx8dj9erVCAQC6NOnT/
ayZcsmVqxY0XXTTTcVjGnXrh169OiRkpGR0fTf//73sC1btuD+++
+PdB8komellItnzJiBQ4cOITExESNGjEDv3r3HCSEQFxdXkDExduxY3HvvvcGJEyce/eKLL976/
PPPm911112Ij48HEeHZZ5/d7HA4ji9durT18OHDERMTg5UrV6Ju3bpewzC6LF26dMzUqVMxe/
Zs9O7dGz/
99FPkZ3IfM6vUMzMApAB4wNphGEYv0zSnIbpS6UqXyzUuGAx+CkX8DTN3zc7OPgRAx05ozknory/
RaM5dZs6cyTExMR/dfPPN7axfmwjpB7QHsBJAPUvfTgCVEK0YuQ7AR8Wcq3eNiYn53u/
3b1HM9xSALgiLRhUacwQAK+I2fjUMo00wGHyDFGXKOVTzY0MkqPMUeRghQ+geS/
v3AF4AYC20BYTkvp1QHPkw8wNCiE+llBsiGR2zZ8/GAw88gB9//
HFUrVq1nrdmeoQDH9sBUMVadAfQDBaJcWb+CaGMjpmKMe0QqvehqlI6gYjGIFovJF8I8bQqkydcu6M
TgDstXZ87nc7uHo9nKRF1UqzjCyHE5IULF14NYOTIkSN1pVHNOYf2UGjKLW+99VYsAFSrVi1JYUyAm
bsjtNFbN/
+DABgWYwLADiJ6S2VMABjtcrm+8vv9HynmmwfgEkQbE7lEFFAYE24iutE0zYdVxgSAnUT0fUmMibCE
uA/
RxsQBAM9CYUww82IiyobCmAAwo0KFCguklB8pdDxQuXLlqQpj4j9CiNpQGxOjEFIktcqfHwPwhMqYI
KK7ESodrioN/
whCR1UNFH3PSSmjxlCojkpLRBsT25i5p9frnaUyJph5N4D2Usp1NWvWvEIhfKXRnBNog0JTbqlfv34
sABiGofoHvp8QoiXCWg8RmDkToXiL5pb2dACPMXOURDcRPe9yuV73eDzLAVxh6f6amf2IDvoDQm73u
pb75AkhOjFzNyjc8QD+i1BJcVXpcCXMvJaI1rOlbDeHKpg+Wsxxz/
qwuFdUCi2HCpI9nJ+fv8qq2XDw4MENAKA4JnqbiH4pRt9jKjPvB/CuZUwuEQ0joumKMVOY2Yuw/
oNl3GRmbkFEVynGPQdguMLYeQshj+5oS/
svdru9CxE9hWhDAwD2EVGTyF8SExN7AYjSuNBozgW0QaEptzRp0mQWENIGsPAcEe1i5petHeHUQqsQ
Vg4RDQPwpuLrc7nT6Zzg8XjmWrUmmPkXZl5HRIMVy/
uRiKIyM4ioNzPXAfCatY+Z05j5GShKh5+E7UKIN2DZrMPcB+AVRabCTiHEdFZUPQXwtcvl6uv1eucj
OgVz/
fDhwy9evnw5KlT48zVxqEDXHBTjBTEMYwlCWRhWBoX1QqpZxiwxDGMZQlkxVl4HgGKOieYwcw+FcNU
nRPStwtg5LoToFAgEhgIYo5hvLyxxFuEAYLvX6z0lPRCNpiyhszw05RK32z0hEAjcBhQ1KJj5E5fLN
dnr9W5XDFsJi8ciPP7u8Nd9EbEmZt6SkJDQ0+12T0Z02XMvgNeJ6AXrfMz8CxE1V7SPIKJMZv5KsbY
MAOMBzFH0KWHmP4QQ/yelnK+Qmb6PiP4PgFUK/
AgRTTRNc4FizC+GYXTzeDxPEZH1Pe1j5mZ16tRx1alTJHxgR1hI7HPFEr8C8IRpmp8rjk1Ghg0xq6f
oO4fD8XggEPhS8bxziGgTEaliQVYAaGpVKwWwQwix0Go8hQNsuzJzEtSVRX8mohrW9sg7e+utt/
Yq1qDRlGm0h0JTLiGiayP/
uBcyKH50uVy3eb3eeYgOtvwGCmMCwEhmHgdLVH94c+3idruHFyPxPAVqT8JBIrpI0f4sgI0IGTXWuf
IA9CWiaad6Ps+h0uX3MvNbiviRaUQ0FNHHLdlCiBEA5ijSJY8KITpJKfsQ0XjrOABORRrlAcMweoeV
JK2Gy8+GYQwhohWwGGoIbeAtYNGNYObDdru9v9/
vX6YYs6w4LwhCx04xRNTG0n5QCPEflSdGCNFfSlmDFYqbAP4bNoCs5dWRmZm5CgAqV66s4yg05xzao
NCUS1wu160+n+85AJFaE24A3b1e7zOIjpvYi1DQpJVnEMoisFardBuG0YWZ2xNRinUQEU0ioolEVCQ
1k5mPMnOC9XoA7wGYDuATAFWsnYZh9ADwOKKDPYtFCDGGiKbBspEjdPRxLULZFNb7DJJSPmcdw8y5Q
ojOYUMoKmYhfGRivc8JAD1M00xRGFBuIupjmub7iA6anMfMHlgCQZk532az9QoGg7MV861h5onMvNS
6Nmb+mZnTFMGUHgCjpZRR8SPMPIGZ05n5PWsfgEPMnI9oMTMAeHrfvn0rASAQCGiDQnPOoQ0KTbll3
bp1zwJAONXvRgrV4RituLSO4it+UTiY0lryPOIKr8vqUuTTmXkQojfYDCKyK+6zwW633wNgtSLbAwC
6m6bZH9Gy0JGgyiiY+X5mngCLARJOea0Khew4EQ01TfMhlffEMIxbmTlommbUhl0czHwTM98LIMnSn
m8YRjcAr8ASqwJgPTMvDxtC1vX1CQaDDynW/
mVMTMxQhErKWw2howA2W5VRw4Gvd4aPsaxehplE9DGAVQpv0AmEPEwXF/
PYSY0bN+4NAEKI2GKu0WjKLNqg0JRbTNMUAJCenr4FQH1rlkMERcT/
ZgDrwzEGRRBC9GNmU0oZdTQBYDlCHg2r7kFe2DhxWtp/
cTgctwcCgQ+h8BgA6MvMTUlRQyScdcKK9pcB9FHMt4OI0qA41iGiSVLKu4iolWK+YTabbQczr1EEby
ph5juIqK0qGFUIcZdpmvcC6GAZ87MQYgqAtxVTPszMHRWGwTdE1NPn8823GkIckgNfpFoDEd0mpXxU
5ekQQkxj5ihPUfjYaQ1OLiJ2deXKlTsCQK1ataJUNDWaso42KDTlFimlAACn01lNSjnvFIcdEEI8A7
VrfzwRbVfVcGDmrcxcHZYgwjAeRXyB1zCMLn6/fybU3odkIspRfa0j5O3wK1IzPwBwsTVWgJn/
APAl1PoPKVLKyyhcotzCFJfLtdDv939EigqdKjgkJpWLaD0JENF4KWUHRKdfHgIwVEq5yHpMBGAeEV
Ujon9Z2rcRUWdmfgXRR1IgojmI1twAM/
```

```
cBcJdizA4iGiWlXGvNAik03x2K+dKsbQDQpEkTlWGk0ZRptEGhKbdEDIoGDRokn2IwowfAPVLKdxR9
r9nt9oVSys8BWOMg9gM4oQj6A4ADqs2YmbtJKe+DWp/
iQQA7mXmZYlxu2ECoa+naRESCiG60XJ8d3gxV8tMrmdlG0SXZAWCBy+V6zOv1LkT00URxzCCiT5j5f
UXfG1JKg4jGWdq9hmH0JqJURL/
XjUR0hJnvt7TvMgzjxvBcAxDNG1A8LzOPEEI0to5h5j+YuS+H5NKtcttAqHCcqtLsDiLKL9wQCQROS
0uLSknWaMo62qDQlFsiBkVMTMwpCQ0R0XAAr1m//AGstNlsT/
h8vk8RvZFnMPOeYr7wD0AdvHcXheSno7QNADxts9mWEdFHxaxxLxE1LdzGzHvCX8o9FUNeQEg1sgjM
vIWZfyWiIYox610uV1+PxzMd6swXFctiYmKeBLBS4WVYg5Bx8JxlDfkIBW4+gegjmt+Y+TurMRHe/
LuYptmZmf+jWMc7CHkgrDxERL+zRaiLQ5VGb6OQEmeUdgQzzwHQQ3EsthtAGoALCrdHDIq9e/
f+rFiDRlOm0QaFptzicDhsAGC323NO4fJ+zPwEojMpttnt9iHBYPDjYoLxNhBRZ2sjh2p0RBkT4bgM
G0IZJFZSbTbb88FgUJntgZBug9Vb4KaQDLeqIupUIopy+wPYT0SfkUK6O5wVcYvb7R4PdQBrFMy8mY
gG5efnr0C0wbUj7H2ISucUQtzFzLchOj00CyEDZJylPZuIuhBRA4TUQq3r+ADATbC8OyKaZBjGUmaO
GiOE6EtEj0IdG7ESQFNYfo5hrY4tCNVmKUJ2dvYn4WtiFPNpNGUabVBoyi2BQMABAAcPHhyIUKnq4n
gUoa9aa/zDfiK6ze/
3r4ClDgcAMPOnUHzBc0i+2+rlAIB5UsotUGslvOt0OscHg8GPoU4P3QXgOkX7BqgloVMRqmVhzWI4g
dBm/
LBi3UcBdBZC3EAKQS4VzLxXCNGVmd8goistfYcBPCalVJWFH87M1VVGDRF9h2ihMBiGcZthGHkIBb9
a+ZqI2sOiT0FELxiGMd00zbUKz9NoZr4ToeJi1vVtARBUxKNkA1gEdTzK+l27dkV0K3SWh+acQxsUm
nKLYRgOANi/f/
8OAI8hpDZpJRWhCpxWw+AEEfXiUA2KqNgIZv6WiDoq2nOJiK0ucgAbmHkGFBLTzLzE5XIN8Hq9i6Aw
XBBSobxAMe47qI85VgBoAks8QDhTYSbUZdyziehGm81W41QDWJk5XQjRmZlHwxKwyMzZhmGMBzDXGr
9CRP8nhPgVisBXhPRCkhT3GkZE35qm+RFCqa+F2cfMlyA6VXdB1apVJwUCgbWwHE0w80sALoX6eOQ3
AJsA3GLtEEJMVQSIgpl/iomJ6S+EYACQUmqDQnPOoQ0KTbkl4na+7rrGnKoAFZVS/
9aZt4DhWtfCNFHSjkNoTRQK/
sRcoVHEQ7Ssx5X7DMM434iWmbVoWDmtQkJCb3dbvdsqL+Uf2fmROumzMx/
WD0C4fZvAWRD4c0IB2c+qFq3YRi9hBDZpmmuVsRARMHMOUKILlLKJgCmWPuFEPeERbKs72I6Ec1nhQ
hVGJXw1xSXyzXPNM0ViA6aPMHMtYnIGiezJlxzZJHiPS0nohwAIxX38gJIVXlOECoOpzIm0oUQ/
Xw+3+omTZr0DrfpIw/
NOYeu5aEptwghajEzEhISFhYTVDebiKLUEIlosJTyMSJSHTFkMLNLIVAFhEptWwtZZQMYEFaFtNYC2
exyuW7xeDwPA1AFR3oB2BSbJUgtgnUAoQDIKClwAPOZeRBZ6nOEGQDgW9M0NxORVdJaiWEYtwohcqW
UixTdD4ZFrawxJIuI6DHTNDcRUZRsdTHMc7lcj7nd7veJKEqQC0AV6zMx8zcul+s2j8eTglBMReG+3
UKIb4sJ6ASAp5n534o5XwRwq/
X9hIM6bwl7ny6rUqXKZQBwKkaZRlPW0B4KTbnF4XBcCgBEZBUpOiqEmAi1iNJUZh4MdbwCwmmWUfER
4SDMatZ2IcTdCKlCRgV7ulyuTl6vtxfUxadyAGSrNBGKWVcmgGdJXVfkM2a+WZU6S0SPu1yud6WUyy
m6cJYSIrrbNM1vAoHACoUWxovM3AmWjAkOFWW7k5nnWbNUTsLnLperv8fjmV5M0KmKXYZhdPV6veNg
KfHOzFlCiFUnMSaeAvCAwhu0mIgSVR4hwzB6EdFoWLRE6tevH3VcotGUdbRBoSmXnDhxwlWrVq1/
AQW1PAAU1KVIZuYUxQa7jJnbQ33MAaCgboUVDykqTwJ4lJmHEFHLwo3MvNswjBszMjKuBFBcvMLviM
6YKBYhxDgA1vLbALCLiK5QeTkAvON0Op/0eDwLAbQ9xVs97HQ65xLRQqsBwsxLiOg8awotM3/
ncrludbvd/
4HFY1AczLxHCNErPOaUsk0A7Hc4HDeYpnkDFEYaEa1UaFpE7jeHmXsh2sO0FcDHAPopxiSbpvkgFFo
Yv//++9enuGaNpsygjzw05ZJgMLja5/
OdBxQtX24Yxt2maU5VuPbXA8ghoquKmXI0gFGIzgTxIDoYEAAWIlTR1BrsuT82Nvb6/
Pz8mgCWqY4gmPmnYlJUi2M0Mw9HdAXOTUT0cTgd1sqXLpdroMfjeQYKBchimO5yuaaGjxKsqZ5biWg
3Qhkzhdv32u32LidOnOhCRI+f4n084XopA61jOFRFVViPFJj5qGEYNwQCgeoIaVFY+Q3qAEww82fhV
NQmlq79RDQJwIJIQzAYBDMjMzPzPrvdPoaILpNSwjRNSCmRn5+fBSA+EAhEyaJrNGUdbVBoyiVENPm
CCy5YevToUUehtqFSynuJqHHha5l5j2EYt9lsNofP5+uh8EI8YhjGd6ZpNreM+1kIsUixYX/
PzJOJaJfl+nTDMG5gZg4LV6mOTraSoqbGSZ7zeafT+brX672QmQu0FMLP1L1q1apet9t9s8Vdv88wj
FvdbvcdAB4CQhtleBz8fv9zNputP4AazAxmhmmaH69du/
aZvXv3PpmTkzOcObRfSikhpTy8Z8+eBxs1avQ6ESHSFwwGj+/Zs+fBI0eONGrXrt1LkT4pJQAgLy/
vC7vd3j7SJqUEM+PgwYNv2+32yeedd16/wu1SSqSnp39MRJWcTucNkTbTNAPp6ek7/H7/
iy6Xq7Xdbo8tfB/
TNIPMfH7k+SLrDl8TDAQCLex2e9VIW6FnrsHMq4t59cq02tjY2LUAehORQ9Wv0ZRllBFYGk15YMWKF
Y+lp6dPNk3zry71ATABsBCigmEYgplBRJBS+k3TzGZmttvtcYZhRI5J2O/3H2PmoBAiNiYmxonQ/
99kXl7e7wCCNputYmxsbCIAMLPMy8s7IKX0AaC4uLi6NpstDvhTXdE0zRNZWVn/
rVSpUj2Hw5EQaQ8Gg5l+v/
9YXFxcQyIquN7n83mzsrJ+Dcd12BwOR3WHw5EopTTz8vKOMDMxsw2A3WazVSYiI2I4aP4+zJxJRD4A
vpiYmCoOhyPeZrMhKytrZzAYnIeQxsesESNGTCjlpWo0ZxTtodCUWw4fPpxywQUXHNq/f/9/
YAmKNAwDRIRgMJjHzAEikgCklDJPCBFvs9limDkQCASOIWRsyEAgcIKIKjkcjmr5+fkHw+53KaWUfr
//mGEYcYFAwA0gEL5eGobhqVSp0kWZmZnbTdP0IiSWFMzNzf3Z5XJdVrFixQa5ubk7hBDS4/
FsA5Cfk5PzVXx8/BVVqlS5DkDWgQMHZubn5+c6nc7WLperKwBkZ2fv9Xq9SwBIIiJmJr/
fT5UrV740Pz//qJTyOIUQACgYDFK9evW6ZWRk/JiVlZXGzITQ0QElJia2qVix4uU+n+/
3tLS0tQAo0u9yuS7Ozs7e7/
```

```
f7fZHr4+LiarpcrmsB2Lxe7+bs7Ow0hIwpIYSwJSQktMrKyvpvfn6+J7K2SH+VKlWa2Gy2Kl6vd5uU
EkREdru9Snx8fCOfz+fNyck5ELm+YsWK9WNjY2swM/Ly8tLy8vLSI/
MRkYiJiTnPNE1fIBDIJSIRbidmppiYGJcQItY0TZ/f788kIiGEcMTExCQiHFvm8/
kypJS+8DjDZrNVEULYAoFADjMHw89TQQhhQ+gHzVJKX+FAVJ/PB5/PF/
nrJUS0k5nXAzh6Rn+ZNRqNRnP24Xa763o8ns/
dbned4q7xeDwPejwelWIlvF6vVVjppHg8HpWMNgAgIyOjU3F9Xq+3qvVeXq83adu2bdnjxo1TqW3+b
TIzMxNOtk4rJ06caOX1ev8PAIYNG7Z90KBBT5zO/Yu7t8fjmeHxeKJqkfxd3G53Xa/X/+7zH43lK1e/
1epMUY65yu93fZGRkqETHNJpygz7y0GjOIZKTk1MBJKWkpFxU2msBgKSkJNtFF130C4DKUsrls2fPH
lzaa9JoNP8MOm1UozlHSE5OHgig19liTABATExMZQAmEbmI6I6I6hQ4d+lZSUpEWdNJpzEG1QaDTnAMn
Jyc0BvJqSkqKSpi416tSpE0/hKFEhRAUianvhhRd+379//8Z/NVaj0ZQttEGh0ZRxbr/
99goAttstlqlvRYF8cxckCIphIAQoklMTMy6YcOGqQaXaTSaMoo2KDSaMo7T6cwJBoOtZ8yYkV3aa
7Hi8/kqM7NhbTcMow6AN0thSRqN5h9CGxVEGDmzJlrSns9/wwySbAAhAEVY2Nj/
OusWbPOquMZjUZZzemgdCo2mjJKcnDwVQL+UlBRV/
ZCzApvNFsfMWFNiVwAsUKI3ikpR+W9to0bJ6Q5n5Rkd2mv40wwYsSIh6Qd4CnNBcaD/ZSUo9mFvrNUqg2
lLIaIPAaxPTU1dXtp+Sd455139gKwDRs2LLm016LRaE4NGcoNGcZYwaNcophHgYwFOlvZZ/
GinlTCJqUdrr0Gg0p482KDSaswzDMPzMwZ+Vwih00c1mjIOlfYCNJp/
6Ctrlz5+L6669H5cqVWVVonjnHR5mpE1IGIyKDSaVsGZZtGtGq1a4f58+eX9rJOC5/
Ph927Ko0ePYujJqNKCNH5Lfu/MmzcPgwcPRKkpKCoQQaN++Sd455139gKwDRs2LLm0l6LRaE4fbVBoNgcXX/
EdHA0l6HRqM5fbRBodGcZYwaNcophHgYwFOlvZZ/
GinlTCJqUdrr0Gg0p482KDSaswzDMPzMwZ+Vwih00c1mjIOlfYCNJp/
6Ctrlz5+L6669H5cqVWVVonjnHR5mpE1IGIyKDSaVsGZZtGtGq1a4f58+eX9rJOC5/
+SQeffRRORyO00Ol3cXWY+5cqVWVVonjnHR5mpE1IGIyKDSaVsGZZtGtGq1a4f58+eX9rJOC5/
```

```
+wzXX389AKBt27ZIS0srNmOidevWAIAtW7agUaNGAEKbXk5OTpHrIptoIBAoaEtPTy9yzbXXXgsAiI
mJOaUMjXr16qFPnz4YN24c0tPTUbNmTdjtdpim+dcPfAb5u+
+uMJUqVcLIkSPxxhtvYPv27QUGBTMjGAzCZgv907Rx40YAQKtWrQAA11xzDQDgq6+
+wq233goAyMnJwZYtWzB8+PASPUfLli0xd+5cpKenFxgsPp+vRHNoNBqNphzBxQRlBoNBbtasGTdp0
oT9fj8zM2/atImFEPzwww/
z+vXrecGCBdymTRtev359wbiOHTty9erVOSUlhdesWcNXXXXUVOxyOIkGZ/8/emYdHVWRt/
D11e0uAJN0BQVDZFFCBcUFcwIERRBEFN3QUFwjpJqLgvn4o4jqu405IAigoKggiiIhARAcQEVFEBFE
QhLCnOyRk6eXW+f7oG0xuVzAZdSKxfs/
DA9S5Vfd0s9xzq855z88//8z88//8wOh4Ozs7O5oqKC33vvPU5LS6uWlMnMPHDgQO07YsSPPnTuX8/
Pz+bbbbuMLL7yQmZn37NnDnTt35rvvvnzJnD8+fP586dO/Npp512cP7zzz/
PDoeD33vvPZ4xYwYHg8ELfufvTpmUWdfvTkrJ/
fv354yMDJ46dSrn5+fzpZdeyl6vl7ds2XIwkRIAX3bZZZkxcUAYhNXSXv/1Ae/
Lfv7553nGjBncq5uA+dhjj7HT6aw+405Z0vTkkDf...
```

```
oM/
d+nS5R6n03mcEKJyzrS1a9dO6dat283Jycn9DcOAEAKVP5eWlv5famrqo1XHre+wjIiS7X+Z8vPzo4
WFhb2EEBdCJ6Rq/gLogEKj+YX2AMDMW4cMGbIrGo3+n/
UQOggzl1555ZWZALpUGU5q3rz5Beeee+5TwWBwOYAkJKLMC2Dm9UR0uWJ8GoBL7ff/
FQoBuAAkPNwAtGfmEsV6gpknAXjBZgsZhnGWEGJ/NBpdjvh2f1X/
HvH5fP8KBoMfE5H987oB+Kyfq5LEzF0V1wPAUwDutQ8y8yKfzzc8FAotRfXvteMZZ5zxLRG9C+BOxX
p9mPkExS5Tctu2bU8gonYAjrPZ+nfs2LEIQH/
FemjcuPGjqnFVMAEAZWVlawB0k5WRoEbTwNFJmRrNL4QBoFGjRumxWGw+EaXY7FuJaBeAIVUHmblcC
DE8GAw+AuDM2t6MmRcAOE9hmgfgPMX9D8VSZi4E0FNxnwoAIKKEc35mvoOIxtgfikR0SUpKyu5oNPo
RgGNt0yakp6ffHwwGZxHRaTX40041WEMwcTeAgGL8M5/PNzQUCr0JIEFbg4guA/
B6DfffT0QJLd6ZeSURLQFwrWLOmwBG1rBeXZkqhAgDgBCCf+1ijaYhoAMKjcaCiMIA0Lp16ywAbara
rIfyR7AFEwAghLgXwMkA/q+292LmxUR0nOIh/
x9mbqd6GB6CdUS0koguUdynnIh21uDDc9ZDuY1tPJCWlrYyFArNB3CSzTbD5/
NlBYPB14moXx18VEJEDwK4EkC6zbRJCDHACtIurOOy2wAkVFow8wYhxN3M/
IzCj2ehCCaYuRTAPsU9Cg9x/
5cAXNisWbMetXdZozn80QGFRmNhGAYDgBCimd1GRFMA+BXTPpNSrjRNs6Y3ZRXLicgF21s8M69nZg8
RHV/
bhZh5F4DXmPk2lZ2IvgKgUtR8h4haIHFHZXx6enpuKBSaAaCHbc5H6enpV4RCoWegCKzqChHlSSmPA
XCKzRQyDON8KeUVRHRrHZddB+Bo+yAzFwAYLqV8i4g8Nj+eZebhNfj4I4CmtrV2IjEAquQVZj4DgNf
pjJ8Sde7cWXUko9E0OHRAodFYHH300X8HcDDhsQqPMfN1NUx7nIjm2x9Sh2AtM28EcLZtfBsR7TvEE
UICVjLgwwCerMH+LoCzFOMriegHAP+0mf7j8/
lGBoPBNwAMsNlW+3y+SwoLC2+vKXipC8y8gJlXE1GGwnwRM7cDkF3HZdcCUAl67SeiKwG8SkTNbbaX
mPlSKHY0ED96qnbUYgUmNelMvMXMyUTUDQAcjniKmtvt7lBYWPh9UVGR8hhIo2ko6IBCo7HYtWvXfA
Cw5dDdC+AaVcDAzLcDeB41P2DsbGXmWUQ01LZOCeIPQ3uQ8WvcDKAmjYO3AJyv8oGIJsGWAMnM24UQ
g4LB4HgAV9tsBS6X64JgMDiAiJ6uo48q1goh/
gXgFbuBiC51Op1B0zRn1XHNNVYQUg3rqOoiAC8S0XE22zuI78K0UcybjMSgai8RyRqSMJcCWFb1z9b
WtO1I0zRV+SMaTYNBV3loNBYVFRVBADBNsxxAEhENY+bRAI5RXP4yEV0DxXFCDRQB+BcRjbcbiGg+g
Cvq4iszjwBwF+LVFHbbq0Q0SJEAGSKiscz8qu36UiLqz8yjAGTZ/
Xa5XOfGYrHORDS9Lj7WwFbDMIabpjlPYbvX6XR+FolEVhJRozqsuZaZj1TNsSpo7kE8x6UqS4mosWI
czDyJiAbaxsqsXJSuivuvAfAvAO9XHazcoQAAIcRFXq93XS0/
j0ZzWKJ3KDSaXwgDQEFBwSyHw9FJSvlPKB44AD5EPJBQ2ZQQ0e3M/
JTCNBt1DCaI6AEiutr+xm3xtrXlniDKRUS3MPNzivErAfydmccp1rswFot5TNOcUxcfa2C/
EOJS0zRzAFTLU2HmyV6v9/lIJPIBFDkQNcHM6wE0IqIjFOYhiB/
rXGAb34p4oqVqB6fy+6uWN0FEq6EOJn42DCMARbVJZUCxa9euCV6v95Nf+SgazWGPDig0GovKKo+ys
rJQNBq9h4gSSjqZeYN1RGF/
SB1q3QxmfsB6I67KPGZWPdQORTYzdwDQS2Gbi7gMdGeFD0OllGOQqIdxD4AUxCsTqsJENAjANinl/
Jq0FupAxDCMflLKsUisHFmUnp6eEQqF3kEdgjRmLiCiCqhLVG9APJ/
iGtuccgCfId791L7eAmZuisTA4T9QlOMCKCKiQbFYbDIUOiOVAUVBQcHaX/
80Gs3hjw4oNBqLSr2GI4444gx7noNl/
4GIvieiwbVdk4hulVKOBNDattZiAN3rkMwJAHOJaAdsD0mL5QC+R+K5PwCMkVJeq9jReEMI8RWAabD
J8BPRbYZhLGfmRYpExjojhBhimmZfAANtpnVWsudzqEOQhvgOw3dQByBPIv5/
m6qx14dITEYFgGUAvieiPrbxr6EI0CwulFI+QEQn1GDfYP1s1GDXaBoUOodCo7EgojAzo1GjRt3sNk
vRcg3UDyMlzPwIM59dmfVfZfxzAEfCtu1v2cpV4k9WZcZ0Zp6quNU669w/
TzHvNQDt7Q9KZl4F4CXTNBcp1DNfTktLyw6FQstrOFapE8x8p2maJUT0iM0UYub+wWDwaiK6uQ7rVR
DRh1YOi902QwixnJlnK6a+X4NWx/
dENI2IXraZfkI8R0V1fDSImfur1qskKSmpEwC4XK7furuj0RwW6B0KjcaiffV2lb057KZ1RPQF6hZM
vCqESCKiS23jG4gorHqrZeYDAFQyzZush7EqmPgZcaXJFxW2D4loPRENs41vcTgcNxLRPHsiIzPP9P
l8N4VCoZmow/FDTTDzZCHEQgCzkLgLcjERnUREdepzQUQTodilYeZ8wzCyVcEEM3/
OzOcoxguIaBwAezCxF0AIioRcZr6NmRvhV4TMKqs82rZtm3mo6zSahoIOKDQaAKFQqFeLFi1GAQkBx
Vor8a8mHQoV7wshvrLKSqtSSETfAPi7ahIRlSsqFYqEECMBTFJM2Q/
gambOtu9qWFoXryNefVB1vBjAdaZpvo3ECpHP0tPTLy8sLJyMuh0/
KGHmRQAeUOVgMHOAiMqY+e06rvlvZrYHSADwDYAxpmm+q5iz3lIlte8U7CeiAGz5I9bR1zIkCm4B8c
BjBeLHRIekMqCIRqP7f+1ajaYhoAMKjQYAM7dyOp0+69eVY4sAFJOiedch1vmCmacy8/
O28XJmnoIaKjqsACDhCEQIcaVpmi8hUa2xAnEBqOeI6CjbtH2GYYxk5hzFekOZ+Vkkym1/
L4QYUFhY+JAqf6SuMPN6j8dzPRHNtcuIM/NzhmEstgKNumgzvAJgsCIw+MkwjEwAsyix/
8k2xJNOE8prieg6S4bbZxt/B4qkTQCzhRD/
JqJaVbxUJmVu3rxZdfyi0TQ4dECh0QDw+XzTKioq3gEOtgR/
AXHBqoR+DMy8XbWG9SY8FsBrCnNuTTLSzLyKiDooTEOklPeo8hisIOcGRX5GhRBiiJQy1/7gJaKbpZ
Qj7HMABIUQlVLX96t8rCNBwzAuCofDE2Cr6AAwz+FwPGya5oewBUm/
wusAzlAET0HDMK6MxWJTKS4lfhBm3g0gRkSt7Isx85VSylFE1MlmepOZBymuX+lyubJM05xbW7+FEE
XWXJ2rpvlLoAMKjcbi888/
fxgAioqKviKiAaSWwf5a8VADM+8gohEApikqN14hIlV3SyCurqhKAr0T8YqNfyjmXEtEJwO4SmEbap
```

```
rmOCQKbuUw82n2UlgrwbEv4qWSCcqVdaVyPdM0R8HW1IuZ13u93iutTq61TvbkeFdWH2xHEMxcwcwX
xmKxl4ioo23afsQbeKn6mNwG4Hzrc1cdzwdwJiU2bNvqdrsHRiKRt6kOfVaklJcBABHpgELzl0AHFB
rNL0QBoEmTJp0BtFfYlwJICCYABC1xqNdg0yNg5neYuQ8UlQIAvoK6JDGbiJywSWBb6z1ERKXMnCC5
zcx3WgqZZ9hMswEEoUhkJKLLpZQeKeXb+B3+PyCiy5m5h71qw9otOC8YDE4hou61XY+ZVyJeDpuQ0y
GEuJKI7qthvbU1JL6+YPk5zDa+0jp+aWObUuRwOPqFw+HHoNb+qIlRRx11VL71ax1QaP4S6IBCo7E4
/fTTK/UlnHablU9hIDGXodx6iD4H29uw9cbrU7w9g5k3IB5k2PuAvG/
Ne0wx51UhxGwpZUJCIDNPtNa6yja+ipn/
g7iAlX1OwOFwbLKqPeqih6HESrR0wFZxwszlhmEMIKI77VUvlr20hiV/
BDCdiEbbDUQ0HPEeHQmtzZn5I6iFqKYT0SKKtyqvyhYA25HYeRVENCgajV5E6iZmNTHVMIxpwWDwXg
ClUPx90mgaIjpy1mgsUlJSMgBl2egyxNtiJ2glCCGuYuY7iOhUm2kdEe2CYpcB8UTBYgD28/vVQoh/
19Dn4oOkpKQx5eXlKxQP//
kAVhBRrm18qxBivBVs2HnU7XbPCYfDq4lItXtSVx4zDONL0zSX2nQtmIiuYOaeAEYp5m0mIlVTr91C
iIcATLHbiOhBAK2YOaEck+PN1xKCFgBLhRBPmaa5pKp/
zFxM8W6xNyjW+ieAxlS3hmhfAdhgmmYhADidTvk7fb8azZ8evUOh0Vhs3rz5eqB6QGG94b+gEl5i5g
AzX4DE7fi9iLe+Vh1ZFAP4SrFNv5WZb5RSvk1EnoEDB8Ln8yE7OxvMvMrhcFxXXl4+m4iOysrKgs/
ng8/nQ6tWrWS7du06PvHEE7nl5eUHF5s8eXLE5/O13rp1a0JeBDO/
5vV6n4hEIguJqOXs2bNx2WWXoU2bNjj11FORmZmJHTt21P6LA6Zt2LBh6rp16xYpylfvIiIXK3qIIC
6Bna7wrwzAKHuljGWbbH1XDylsC6Doz8HM3zkcjkzTNN+1l+USUQ7iMt2wjT/
gcDg2c7wjaW0JEdFzAB6tHHC5XKJVq1ZXBIPBmtQ2NZoGgw4oNBqLzZs3bwEA+Uv/
8tUAhloPnWow8yNE1AxAQGH7F+KdQBMgovdhk59m5hLDMK5EXGuiaSgUwtKlS9GsWTN88MEH5U6ns3
80Gp1QNXmzZcuWmDVrVmjw4MGPXnLJJcc8++yzuPbaX/
I+161bN9e6n9vmwnyfz3dDMBicDaDL6NGjkZGRgVatWuHFF1/EjTfeiO3bt2PcOFWfMCVL3W73HU8+
+eTnfr/f/ib+CoClUso3FN/RYgAtoG79PoyIHoMt74SZFxDRDCuosPMJEZ2i0LvY5XQ6L43FYm/
ak2mZ+TlmVv35vQZgqmma8+pY1jraKsk9iGEYYGaXYRi6dbmmwaOPPDQai3A4LB0OB0Kh0OfMXOZ0O
q+MRqPvw/
bQY+bXhBAbLV2Jalg7GTWVXr4Nxa6FEGJQLBZ7rLKCYPHixRBCICsrq+zhhx927t279w6v13tZ1Tlu
t5t79+59ba9evXKJyNG+fXuMGTMGS5YsQe/evZ/s2LGj6jhglc/
nuyIYDL5GROfMmzcPr7/+OsaNG4dRo0ZVNs6KXnfddSlVdztqgpl/8Hg8V4TD4ZlJSUl2/
Yf3HQ7H87FYbCUAj23eQiJKh6ICw8qNGAngWJtpDRE9zMwfUaJU+DLEk2XtHUyLhRDnx2KxZ5Go+jm
FiC5CXKOiKkucTucd0Wj0UytgrC13A7gVth0Xh8OBffv2fZyamvpFHdbSaA5L9A6FRmPh8XhMAIhEI
sXp6el9Y7HY3YqKiU8Nw3hDFUww8zhmvh4KnQJmXsjMqgTCf1qNuw7KQi9evBinn366DIfD1wJwLl2
69G77vL179+5i5qcrRaPOPfdcAEB+fv4aABe43W67kNMmp9N5figU+hdZzc1ef/11tG/
fHjfddFOlUFYhEaU4HA40afJL5eR3332HrKws9O3bF61bt8b555+PWbNmHXA4HAO+/
fbbV30+35lvvvkmvvvuO/
h8PvTp0+cAM98Qi8UW7t27N3X06NE48cQT0aVLF2RmZm6PRqMCwCn33XcfTj75cf+f8smXLnvJ6vRP/
+c9/HqymePfdd+Hz+fDll1/eUl5ePvO2225LvuiiiI9C6dWuceeaZuP/++wtM02xkmmb7448/HqNH/
5K/SUQDFixY8IjP57tg7dpfGn5yvKtoR9gqeZj5B4fDcXksFptZl/
JQxAPFv0GhrFlWVra+pKRkXx3W0mgOW3RAodFYSClNACAiKiwsvB3AnbZL1iEu76xSPpyGeN8HlVzz
GiI6WXF+/
wARtalawmiaJhYtWgTTNN+69957C1u2bInFixdXW2z9+vVfpKamplUVZaqUeU5KSlK1Lw8KIfrFYrF
MADcC8TyRTz/9FGeeeSasN/51Kn0NADjyyCNx0kknYdSoUZgwYQI6deoEv9/v/
uKLL+475phj+s2ZMwd9+/ZZF27Zt8c477+xq27btJYj37jhm6NChmDt3LjIzM3H99ddvLokW/
cuHF9AODkk0/G1q1bEQqFACA7Pz+/VVpaGr7+
+utfvrg1SvEVAAAAIABJREFUayKNGzf+4ZRTTTslOSkpq3rVrV1x33XWYNGkSPbwYsCM7O7vlrFmzTj
IMA/369cPixYsc2CuAnDWt99+e2GTJk1w/
PHx+ICZvySiEBGdbvuYhYZhnBeLxZ5HDdLoKixRshVQJ9/+X0VFxWatQ6H5q6ADCo3GojKg8Hg8Pnt
mPzPvNgwjg5ln2M/pEU8uXGrXNrDYgrgok73cdCaAb2HrtfH555+jsLAQTZs2ncJ/
bu3RsfffRR1Uuebtq06rRbhGE8w878V5mkul2tS8+bNz3C5XAmy3xpNQ0QHFBqNhWEYEgBSUlKqnbczc5kQ4spYLDaZiJ
rbpv0E4GGoVSb3WhULR9vGPxNCvKTSk3j55ZeXAiibPHnyOADeHj16YNeuXXYNeuXVi9ejWYeRaAcLNmzVpX
Xl9cXIy33nqr/
V3+5XrVqlV9K2QK2TqWVfSZ27969kYj6AcDgwYMxZ84c3HTTTf/
lJSbVrCgoKtsRisf8DMAgAli5dCgA466yz8Pzzz+Pmm2/
fc2aNTj/
BUqIYryCikVLK8QlfFvANM18UiURWejyedJ/P17uoqKjW/
WA0msMVHVBoNBYnnnhiCgAIIYyq40TkZ+axduVFqwT0Oma26z9Ulj5uU6g1bjMM42ZmnqnQk5g6f/
```

```
78Rj169Ch3Op3HAkCfPn0AALNnz94hhFgC64G4adMm+Hw+tGnTBiNHjmzSrVu35JdftnfgBkpLS0sQ
P36oRmpqKpKSkmJSyoMaEM2bN0fPnj3Rvn11kdC8vDxkZWWhd+/
eyydOnHjNnj17Eu6zevXqJampqWkADpaaFBcXAwC6d+/
exefzTfP5fOjSpQsAYMeOHWDmHwB83Lp160xmTv7yyy/RtWtXdOnSBW++
+eaitWvXnldSUuI+6aR4O5A1a9bgwgsvBDN/
tmDBguRgMIizzz77oA8ul2tecnLyIrfbPQwA8vPzYRgG+vfvDwAPqUpNAYxh5n015MSUIp5XYpfiBo
DrEQ8k7fkyXwNoYs052qryMKSUM4qKihLap2s0DQl9tqfRWLRr1+61b775pjKnoJKXENeZUPXUuALA
w6RoPgVgPYBqYlfMfICIhsRisTeIyGezLUpPT78XwPZly5bB56u+Sz5jxoyyhx566IXK37ds2RLZ2d
koLi6e36NHj/6pqdWrL7/++usZAAb36tXrNcURDZh5Vps2bQauXr36V/8PePnll3H66afvu//+
+282TfNj0zTtl4xKS0vLNAyjWoVDWlraXgDNpk2btr9x48bVHDzuuOP2OZ3OgcOGDVuYn5+fvHHjxs
pxNGnSZNnXX3/dYtu2bX09Hs/BIGT8+PFRABXTp0/v5vFUj8WsXIaRY8eOXXnbbbcZ+/btQ35+Ps4+
+2ykpqbeL6W8XRHATXE4HDNjsdgK1ecmojUAzlKY7gZwOeKJmFVZh3gPkYMN0SpzW5h5VlpaWj40mg
aM3qHQaACEQqE2sLL+qwQUXwE4APW5epYQ4lwAvRW2NQrlTAghrgXwL0psjPUtgMsCgUAuAEydOhVz
5szBnDlzMHv27NIuXbos2b1797G7d+8+OCEpKQk9e/ac279//172YALAS/
Pnz19u3VN1fj8PwImXXXaZY+PGjZg1K2ED4yDMjB07dvC2bds+Mk3zfSJq/
M0331S95CkiMjp06PC3qoEGM+/evHnzKABo1KhRas+ePVH5o0ePHqUtWrToF41GH/nHP/
5x1Pr167Fq1SqcdtppAPBhSkrKh8XFxZ3XrVuH7t27QwgBZq5YtmzZ2rZt2yZ5PJ7GQPzIZdOmTQiH
w+UALgHw7sUXX9zcMAwsXrwYixcvRseOHT+WUg4hWydSAJ+53e57YrHYfKi1MN6EOpiYBsBRWSlT5f
PuIKLPYOv3QUQIh8NlPp8voY+KRtPQ0AGFRgPA6/
VuCYVC3YH4Q4CZDzDzXCh6YAB4mZmLmPl2he0nJL65gplvl1IOge0hxcx7mLm/EOKybdu29e/
SpQsGDBhw8OF79tln3/b444+fAgCLFi06OC8cDlcA6KHYfXjP6/
Xed+ONNyYoe1p8CKAREXUcOXIkunXrhszMTIwePRrz5s3D008/jaeeeurgxUS0Nykp6StmviI/
P7/5jBkzMHr0aKSmpmL+/Pk/RyKRjcz83AknnIBNmzZh/vz5ePfddyvWrl171SuvvDKyf//
+uOOOO7BgwQL85z//wZgxY9C5c+dvAQwkostOOukkxGIxvPHGG2jbtu0uZn5+1KhRD+/YsQOLFy/
G3/4W/yoNw7hmwIABR23YsMHx9ttvY/78+cjKysKBAwf4008/3bVnz55sAKekpKSgd+/
eGD9+PHbu3ImMjAyDbC3KmfkHl8t1aTgcnonEZmCVfVFUnVyXM/N7qKKEabGfiF6pQQq8qLi4eDMR/
bqwh0ZzmKOPPDQai08+
+WQvABw4cGADEb2IeBKfnSUAJgBI2CZn5gIAR9iFl4goD8ARiG+TV73+ABH1ZebOpmlOWrJkCUaOHF
nVficR3XnmmWem+Hw+LF68GEOGDEF5eXmJpbxoPzZZ4fP5hoRCoRkpKSltFL4v53gzs/
MBwOPx4N1338Wdd96JFStWtW0aLjd7q/vueeexlLKXAAYOnQoNm7ciBtuuAGE+Y+UEASElJQdeuXfH555+jTZs2rwKYmZKSgg4dOmDlypXIy
sqCEOJy0zSzxo4de4QQAg899BAcDgcuuugi9OjRY/Abb7zxcQ4Xbt29/
79u3tJaAhh8MxIBwOv0RECc3AADxBRCr1zO1E9CgRJfRZsQLL5xSCW4Xbt29/
yzTNvnaDRtMQSfgXoNH8lZkwYQI7nc4FgwYNOpOI7CqK24jobGb+FHHNiaoEmdlpT+Bj5kVCiDdZ0a
BLCNFHSrmHmVcoNCqelVKeZRfWYuZdRFSBxDfrTUTUjZlfhKJNOYCvmPkTIrql5k9fHSK6hJlHILE/
xteWauVMxbRrmLk7qTuEPghgCTMvqTrOzBVCiBFSyhcV3/mNzHyKpaBpX+8yAEfb+4REIpFN7du3P/
L++
+9PDgQC9jm9pZTnENED9vWsdS4gog628TLDMK6XUuYhUTU1k4juRaJIVrlhGD3ffvvtgQCGjRgxojU
0mgaOPvLQaCwmTJjgBIBmzZqdbX+wWUqSA5j5JdiCCcvGimqAbwzDmKAKJohoqJTyR2ZeZA8mmHmml
LKNIpgoBVCCxGAiSER9pZS3QRFMMPNGAG/
XMZi4mZkvQWIw8RMRjZNSTlVMu5eZm6uCCWae5HA4pltHBvZ7jZVSPqUIJp5m5uQagolbpZROezABI
JiRkfFzcnJy8uDBg+1zhjFzK1UwAWA8gBPtwYTFzVLKZ5EYTNwOYDBswQQACCEuAND0uOOOu14IYf9
cGk2DRAcUGo1F9+7dGwGAYRgJVRGGYVxDRH0AJMhnE1EJbD0cmHkngHtN03zNfj0zjyOiORxvdtXcZ
ltJRDtJ3YK7wJ7QycylQoh+zHweESX0ELH8eAo2Aa1fYbyUsi2A62zjQSK6XUo51Z67wcwTmXkTET2
j8OEjj8dzfzQaVSVA5jDzKCI6wjbnHSHE50T0FBIZj3ji61tVBxcuXIgOHTpEPv/
8839MmjQJXm+13mJPIp7fktCoDMA0ZvYQ0bl2g6VmegsStUQmENERRHSeYs4gKaVbSrnA7Xa3EUKkF
RYW3qq4r0bToNABhUZj0apVqzcAQIjq/
yyY+WEAP9WgiLgPiU2pDljaFZMVSZNv+Hy+f0kp59mTBQH8JIR4H0CCshQz/
6B6eyaiK6SULQFkK3wrJKLbACS0AbfWLFYMvw9gi303g5lLmTmLmScQUWP7HCHEG0Q0XbHeVy6X65p
wODwXgH3bfwmAvpQo+b0UwL+llDMU680RQuRJKd9X2J4cO3ZsyvLly9GjR4+4+q43MBTK1hzhxLjjtB5
ZSZJzFzHwAn2sY/
YuaPES8frQYRXS+l3GMpckIIgVgsBiJ6NhQKqZRUNZoGg07K1GgABIPBR2Kx2AVAQkDxvsvlejEaja
5UTNsHRSMwwzCGSSkfs791A/jY5/
NdEwwG3wVgTwjcD2A8Mz+puM82RakpAIxk5qBVrmhnLxFlSSlfqUGHosCun2E9WD+AQvWTiEYAeByJ
wdOXDofjkVgspuoCutXlcvWPRCKTkNjjZCuAdkg8PtpARDcQUYJmAzN/
6XQ6745Go0sUOyQP9e3b9yrFZ10jhLjFNM1P7YEQM38khJgM4F37vQDMQXw3pZdtznorf2SBYs4NUs
pVRLSscqDyOzFNc0mzZs1Ubdc1mgaD3qHQaAAw8wQp5Q9ANTGiDU6nMyMSiXwAW96CJcecEEwAyDJN
MxNAV9v13zHzxcFgMBvAxYr7P1JZ+WAb34XE7XYQ0TOGYSxSVR0gHpxcIqX8l/
1IxVpzoz2YALBFCPEyFMEEM/
sRb5RmzxXYIoTIjMVisxX5DyHDMM6NRCIPwHZMZO12eJAYTOx2Op2XMPPbsAUuAH5yOp3XxGKxuYrP
lE1EvRTHQTsNwxgspZytCJ5WOByOB6SUrys+7wpm3mclfVZln7Xz9KYioLnd2pVYDCDNvuZ3331nPz
```

```
7SaBocOqDQaACkp6dv27Rp07nAwR2KIsMwBkSj0Rwi6lb1WmYuViRgAsATzHya/
VydmXclJSX1I6JRAEbYJzHzQ0R0l+IhVUqJgkxg5plOp/
Np0zQXI7F0tJSZL0BcwVO1q7FJcXSy3wpoJinudRuAK5GorbHPMIwrpJRv2X3keLOtAVLK/
gBG2uaBiBopckcOCCH6x2KxZylRrjxERJdGo9EpAI61zZsFIAWJOwllAC6NxWLjAXSx2VY5HI5rTdO
coUiI/YGIlhBRhmK9661jrGpHNET0oMvlet1KsLX/
eRUDwLfffhu1fw8aTUNDBxQajcX69etLAGDHjh0ThRDtmPlGKHYTFG/
jYOYZiAscDbeNVwghLigvL78AwCOK245H/
IFtP0qoUPnIzCtdLtfISCTyIRQ7F4ZhXExEd0ItFb6JmRMCFAD3AUjID7EqKM4kor628QoAF5ummUt
WF9CqCCGuJCJvDTknKkzDMAYx8zAA/W22KDNfIKV8hohOs/
mxkoi+h6J1uBBiCBGNtRJpq85Z73a7L4rFYm8i8fsrJKJXoRYzGwrgSUWQlp2WlvZsNBrNR2LpaOnW
rVtfsPyxy35rNA0OnUOh0ViYpikA4MCBAz9JKa8CcFstp34mhHifmRMqOohosJTyWCLKUcz7gJm7qR
7KlNh3AgA2u93uQZFI5C0o1Dit6oKhUB+pbCSiJMUb+TgAYxU7LnOIKIJ4WWQ1hBBXSSnHElGCDwBG
EVGBlHIpEdXqhYWIMqSUHQCMUpiHAcgiomqNtaxciymI91qxr3e/
lPJasgS8qsz53uVy9Q6Hw68odp1KrQqVxxTrjQZwC2zJmQDe93q9d4ZCoQUKGwzDuCQUCrWnuPKqW/
nhNZoGhN6h0GgspJQCAJo3b94eapVMFZsAPK4KJhBPmozVVP2AuAT2aXYDM6tkmoscDscFkUjkWah3
H4Ywc3co+o5wXOWxFLY3cksf45+Kks1VzPwJgLsU98myemOoSiyfATBHSjm/
hoAoASK6n4h2IL5Tk2ADcAIRXW/
zbxcRPQpFMAHgTWburCi73eTxeP4RjUZHK3IjIIR4gpkTWpsz80PMfA4SJdNXMfOQUCg0w26zfH/
GNM2HW7Ro0c5aX+9QaBo8eodCo7GQUgohBFq1alXb8r4g4m/
V0+wGZn4ecUXJ5Yp5WwFsB3CRalEiSlKMXRyLxTKh7jExkojcNTwQy4noJwBn20z/AfA3+
+4IMxcIIV6sIUB6lJnPIKLLFbY5zPwwx1U/
7QmVNfG6EOLtWCy2SlEh8raUcifFZcur+ldCRH4ACe3GAawG0BiJ3+s2Zv5HRUXF34ko4TsiorHMfL
Ni9+ZVKzCy7/hsdblcA6zqFbvwF6xclOuIqNlRRx118u7du/
UOheYvgd6h0GgsTjvttHYA4HK5ajtlKIAXkZjVP0cI8SoRzbdPYOYyZv4SNQQTNXANM3cBcIfCdg+A
raqESotPYQsmmHkDAGnf9kc8B+QeKWXCbgGAKczMRDRUYVvm9XqHMPNMStTWqIkPHA7HraZpfqDISf
kYwGx7MAEAQojrmPkZAF6b6WcAu5H4ve4TQvQ2DOM4sglhWWQzcwZswmQA3hZCfIrEXZqQw+HoF4vF
HlXtdACYTkS9YeXEEJELADp16qTqG6LRNCj0DoVGY1FSUhIDAKfTWZvLhzDzGCKyJ+J9aRjGOCnlR1
C3xV6g2I4/
FI8yc5TizcrsPCmEWGRpLKjmfmqvOEG89HEnFMcmRHQrMz+rqDZZhLgypUoF8wfDMC4KhUIv2xMgD8
FqZr4qGo2+S0T2qo0fhBAvMLNKG2KIlHKUvUrFqsDYTUT9bePlhmH0NwzDHY1GE3q0WxUivYjILrj1
gSWetVAxZ0AsFvMDSOgsiriA1rGI62sA+EWHIhaL1TpK1WgOV3RAodFY/
PTTT0EAMAzjewAJiZJVuJuZLyWi7lUHrVyFW0zTnKfa9mfmj4joktr6w8wzDMPIl1IuVpiziegVKeU
X9gCgyvzTFIHGWiiCCWYOIF7tYff7Wyvh1N4zA4i//
Z9rmubNRFRbnYWfHQ5Hv2g0+pw90dJa7w4p5ZsKv29g5r6KOaB4a/
CEXBQiulwIsSUaja5CYh+ORUTUBsDxtmnLhRCjpZSfK9a7lIj6MLNqp6hSzOoU2xwAwE8//
bReMUejaVDoIw+NxqLyLfLbb7+9GvHz+ASYeSLFezhcZhsvJqIbmPmdGrQjVhFRv9r6wswrHQ7HU6Z
pJjTTAjDNMIz/
k1IuQGIAcBB7LgYzr4I6ofMxxEtXj7WNbwPwuiqYYOYyIcT5zHw2EY39tc9jETQMo28sFruSbFLXVi
LqKGbOUwRIjxKRyz6nCvbjChDRrV6v9+NoNDoPNslvZv6CiNJhe/
gz8wohxIVSytcUa94L4BiOy7DDNu87xJNsE46xCgoKngEAKaVOytQ0ePQOhUZjIYRwMTPatWt3JDOf
YH9Ltt5qv2RmlTT1Dcw8ntTKlLsAqLpYKmHm7YZh3BCLxT4ghVy0z+fzB4PBj1XlpodYc5ciZwIA3g
SQrtBrKCGi11FDUzEiuoKZmwBQdR2tyYcBVt+RhAoaivcceRS2AImI8ojoE+sIqba85PV6nyssLJxp
30UC8COAJgDsuR6rfT5fv2Aw+BAR9bDZcpm5EICq9LfQ+p5U5aaDtm/fnqLLRjV/
FXRAodFYSClbEhGaN2/+hqLscS3iDxRVCei9AB6hxCZXlQ24GtsDg5qwqjKuklJOVwQny30+38XBYP
ANxYPykNSwa7ICwJdE9LTi+jcR/1wqbnI6nZuj0aiqh0hN9x/EzAXM/
JXCNpaZs1Al98BitmEYT8disYTjh0Mwx+fzjQoGgy8DqJarwsxlROQjIp9tzo9Op/
P8oqKiDEps8T6fiJYCUHWNLbN2Z1RaGEMBfMrMA4gIhmHogELT4NFHHhqNRePGjU8GAMMw7OftBcx8
F4CE5k7MPNmqEmhbw7KO2gYTAEBEmYi32rb3zVjj9XrPD4VCY+uSh3EINgkhnlAFE1ZCZEA1iZn/
7XQ6p1tHCaqkUxVZaWlpC4loLhKPEnKY+XQkCnX9x+VyZUWj0comXb8KM6/
yer1XFRYWPgi15HcyEqXKC5j5nEgk0tt+tMPMG5l5Vg0ltCCiuwA8qDA9LqVsysyh44477mxAH3lo/
hrogEKjAVBcXJzeokWL24FfEumAg7kRfgCTFToF+UTUndQ9M2CtpUyYVMHxfhqXwNaJlJk3OByOvqF
QaAgULbP/C/YT0XBmzlX4kH+IgGWuz+d7wGqWVlMAZV/vYZ/
PNyEYDL4BW9DAzAuYOR3ABbZpXzPzhZFIJMde0XEIfnK5XBeEQqGMOuR07HM6nX0AHEVq8bEPiCjhO
7J8fwhAFmwN4qzS2c8rA7WmTZuOAIDmzZt3rqVPGs1hiw4oNBoAsVjswYqKCi9QvX25EGIYgKfsRwa
WgFMeFJLLln1DHV2YQ0RpCtGonwGcG4vFTodCTfK/
QQgxmJkfQ2K31Gk+n28ggC2KaV8z8zWhUOjNGnIxEmDmyenp6Q+EQqFnFEHKGiJSdfT80el09iOiaw
EMrOV9SpxO54BoNNoPcV2QWs0hon6maYaJaI7ikp8Uxx+VvAngDACdASAajSIcDqO0tHTNDz/
8MCkUCr0eCoWwb98+RKPxnmD79u07tTZ+aTSHM8ridY3mrwYzJ33zzTcfr1ix4vTBgwdXjgUAXEGJz
bE2Op3Os1JSUgqDweBc2NpzI16VsMdSy6zKJrfb/
fdwODweVR6WzLwewBkUb39ddXdir2EYPZg51TTNT2rY7ZiHeKt0u6z2u6qdBmYeIYTYxczv2cZXpKe
nnwkAhYWFw6uKSjHzzl27dv3dMIzk1NTUxdZ2PqSUsH7ez8ypVccqKipWLF+
+fMwJJ5zQp2nTpvcyMyp/mKa5b8OGDfd6PJ7mrVu3vh+A2xrf/
```

```
+OPPz5uGEZy27ZtH2BmAICUEgAQCoU+dDgcTZKTk3tUjhMRCgoK3gLgbtas2SVSSlT9UVxc/
HVSUlJXZhZV7i8PHDiwPhwOl7vd7vSkpKS2lfdgZsRisWgsFitzOp3VPpN1T1nZ86U2tGnTBlu2bMH
OnTvLCwoKvsvNze02fPjw8ydOnPhhbdfQaA4XdECh0VhkZ2dfahjGdABU2dfDjhACUsoKAJKZpWEYD
iGEp/
KYhJmj0Wi02DAMl9PpbFLl+IQrKir2IN49kz0ej9cwjMbMHKmoqNjFzFEASE5ObmUYhoeZZVlZ2U9S
ygq3293U4/E0B345jiEixGKx4pKSku+FEI6UlJROQogkIkI4HN5dVlb2s9fr7SqEcJumCQAoLy/
fW15evguAw+l0prtcriOsh6VZUVERYmYDgJOIHAAa/Jm/
lQAbIyI4HI7GQggSQqCiomKPlLIsKSnJ53Q6U4CDu1ZmSUnJuiZNmhzjcDjShBAgIhCRGQwGP45Go3
u8Xm+btLS0UxwOh8cwDDDzku+///4kZh6/cuXKvEmTJm32+/
2ziGgggMtzcnJm1+uXoNH8juiAQqOxmDRpUjPTNDsBGGGa5jEABBFRcnJya4/
H08o0zYoDBw78GIvFKsLhcBuXy1UghIgZhpHcpEmTY5nZ3L9//1oAUQBwuVxNmjRpcoIQwti/f/
+aSCQSZGYmIgkAbrc7taKiIlj5e8TlsEV6evqpJSUlG8Ph8D5mlkRker3ejk2aNOlKRDBNcz8A3rlz
52zTNMsBmE6n09myZcsLotHo3p07d+ZLKaMul8vVpk2bKxwOx9HhcHjT5s2b34L1b56ZKT09vVPTpk
0vKCgoeKO0tHSf5QMxMwFAamrq0UIIdygU2kzxSIasz9XoyCOPHCCljPz8889ziYicTqcnJSXl2EaN
Gh1bUFAw3zTNWOV9iIhatmzZz+PxtN27d+
+84uLiHUR08D5NmjQ5mpkjpaWleyuvBwCv13tiSkpKt2g0Wrhr166PotGoQwgRISJKS0vraBhGamFh
4apKvw3DcDdr1qy3w+FIKykpWVdUVLS+8j5NmzY91e12tywqKvqyrKxsV9X7MDM5HA53cnLy0aWlpd
ullJFKe1pa2olut/sYACguLv6qvLx8LzOT1+vt4na7WzAz9u/f/
7X150gAQESO1NTUzswcC4VCFxLRYCLaPGLEiINHVpmZmWcAOCovL++dQCCwk5lXSymvnThxYvB3/
mut0fzP0AGFRlMLgsHgTUT0/t133/0ogCsAfLVx48azlixZEgOA0tLSI03TTElJSfm+6rxQKHQSM5/
o8/ne+K0+hEKhDCnlWenp6ZnFxcXpKSkphbWZV1hYOM7n8z1NRCWKz5Xq8/
n2/1bfakMoFPq71+v9tC5zgsFgZ4/HExo1atRDQoi/
A1gTi8XmSSlXvvbaa+vs1xcVFXlN0+ybnp4+w24rLCxMSU9PL66r3/v37z/
WNM3LfD7fEzbfXmLmr9PT0xN6jtTlXhkZGR0dDsfDzDxXCLGDmUcDmKx3LzSHGzqg0Gh+hSFDhhzVq
FGjXADnAthQUVFx7ZQpUxL0FDR/HEOGDOmTnJw8A/
GmYEXMXAAgKKX8kJlXTJ48Ob+eXfxdGDVqlLuiouJ5IrrIKlUWUkpMnDix1gJiGk19oQMKjaYG/H7/
ECK6G/Fs/gXRaDQwefLkbfXt118Vv9+/
hpm7VlapwSaYAAAgAElEQVThWImSYWbeAiCcl5dn17I4rBk5cmTjSCTSXQgxEUATIUQ3KWWTnJyctf
Xtm0ajQgcUGo2NQCDwIoDBABpLKafk5eUliCRp/
vdkZmY+CuBeIUS1/7eYGbm5uQ36/7Jhw4YdbRhGhIgyAAxg5m+SkpJuffHFF8P17ZtGU4mW3tZoAPj
9/pOI6HkAPQBsN01zzMSJExPOxjX1RyQSedvlcl0PoFXVcWbeWE8u/
c+osjP2eCAQWEBEM8Ph8DEALgwEAndJKafl5eVtr08fNZoGHdVrNLXB7/dvJKL2AP4TDoeHv/
baa5vq2yeNGr/fnw/
gH5Xls1JKTkpKSvqLvqkTAA4EAm8CuJKILp8wYcKs6667Ln3KlCm1StjVaH5P9A6F5i/
J0KFDWzidzieI6CoA3wLw5OTkROvbL82hicVibzkcjrMQF8P6iYjuCYfDhQBq3S+lAcEAkJOTcxWAq
zIzM9sCgMfjWRcIBER5eXnnqVOn7q28TqP5o9E7FJq/
FMOHD+9hGMZTADoxczEzj8nLy3u9vv3S1J7hw4dvAmBMnDixDQBkZmYeL4RYkZOTU9tmZQ0ev98/
xjTNGYZhMBEtApCbk5PzcH37pWnY6IBC85cgMzPzKiHEHQBOIKIvo9HoLZMmTVr1qxM1hwWZmZnnCC
Gm5OTkJLSQ/6vj9/
vHEdGFOTk5pwYCgQHMfHJubu4j9e2XpuGhAwpNgyYjI+M2h8MxDkAygGkA7srJydlZz25p/
gD8fn8GET2Qk5PTpr59+bNi7eZ8AqAiJyfnmOHDh3eeOHHit/
Xtl6ZhoAMKTYMjEAikSikfF0JkMfNOIprcsmXLBx588EH567M1hzPDhw+/WgjxVG5ubqtfv/
qvS1ZWVqvs7OwCv98/noiGMvPg3Nzc9+vbL83hjQ4oNA2GQCBwJIB/
A7iYiA4w8305OTk59e2X5n9LZmbmo0KIrJycnPT69uVwwO/3DyQid05OzoxAILCemd/
Nzc29r7790hx+6IBCc9iTlZXVJhaLvSCEOBfAPma+MTc3d059+6WpP/x+/70AMnJzc4+rb18OJ/x+/
ygiulEI0ScSiTQ1DOPY3NzcmfXtl+bwQAcUmsOWYcOGHe1wOF4hom4ASpn5Fr1tq6kkEAjkMnPX3Nz
c0+vbl8OR6667rkNSUtLXzLw+Jyfn1AcffFDoY0PNodABheawY9iwYc2cTmcOgPMAFEgpR+bl5SwY2
780fz4CgcBKALtycnIG1rcvhysZGRl9J02atGjEiBGzmLkDM1+Zm5ub0OlVoxH17YBGU1sCgYDT7/
dPdjqdmwA0J6JLc3JyjtPBhKYmcnJyujNzJ7/ff3d9+3K4MmnSpEUAEAwGBzPzNwBuBgC/
359Vr45p/nToHQrNYUEgEPg/
AA8B2MvMN+Tm5r5b3z5pDh8CgYAZiURavfrqq7vq25eGQiAQmAngEiI6d8KECYvtzV/
aqwW4rmIJ1sOz83NfbW+fdL8P3tnHt5E1f3x7tHpVaaAMsOsoM/
RRCUVQVFRVQWQaBsL8jWpLyIigiIK4iIC6KIG00ooOCCLAooILKbCr4oqiIKCAia2lKN2ibmXt+fy
StzWSKVMEUej/
P0wd679x7T6YzmTPnnuXig5mdDodjN4C4cMtyqeB2u3sNHTq0FhFdBgAul2s7gCS3262K6pVSlIVCU
SJJSEhoIYRYBKA2gKlut/
vJcMukuLhxOp2fAcj0eDw9wy3LpYjT6XyciMYYhtHa4XCkvvnmm2nhlknx76IUCkWJw+l0LiOiDgCW
RERE/LeUVpJUXABcLpfLpPJ7T4t4ZblUuW+++6LyMnJqSOE+JGZX3K73cp/
pZSgtjwUJYZhw4YNsdlsc5h5n2EYN6iUwIrzDTNPIKK3ADQOtyyXKoEXgL1Dhw5tY7fbHwIAl1uVAO
Ajt9t9MrzSKS4kykKhCDtDhgyp5nA4NjBzrBBibFJSkqr+qbhguFyudGZ+0uPxvBJuWUoLTqfTSURJ
AEao7LWXLipsVBFWEhISHnU4HL8x836Px1NNKROKC41hGOMBjAq3HKUJj8fjcbvdkAkiauTk
xMbBZuuRTnF2WhUISF+Pj4MjExMd8SEUkpR8yePXt9uGVSlB5cLlcegP+43e5F4ZalNOJ0OqcT0b2a
```

```
plX/
7bffTq9atUr5SV0CKIVC8a8zZMiQng6H4wMAn7jd7u7hlkdR+khISJgthGjkdrtvCrcspRmXy1XWMI
xemqaNzcvLu2nevHmnwi2T4u9T6hQKZu4NQL2VlF56EdHScAuh+Ocwc1kA2eGWI0xsI6Lrwy3E+cLl
cn1ARLWSkpKui4+P1xYtWmSEWyZF8VFRHgqFQqEIK263u0/+/2NjY/c7nc5vVL6Qiw/
llKlQKBSKEkNERMTlRFTD5XJdMWzYsPrhlkdx7iiFQlHiyc3Nhcfjwa5du8ItikKhuMC8+uqruW63u
21aWtovRHS5y+VKczqdV4VbLsVfoxSKf8D06dPx4osvFvyek5ODe+
+9FytXrvzXZTlx4gTmzJmDkycvvbwx27Ztg8vlwrhx48ItikIRVlasWIGNGzeGW4x/
hUWLFhnJycmrmflJIno03PIoFCEwc28OsHv3bh4yZAhXr16dq1Wrxh07duRVq1bxudKqVSseOHBgwe
9paWkMgJOSkv5y7I8//sj33HMP16lTh2vUqMEDBgzgHTt2nPPaZr7//nsGwB9//
PHfnuNs6LrOM2fO5A4dOnB0dDQ3bdqUJ0yYwKdPn74g6xUmJyeHX3zxRd6yZcv5mE7ty14iMHNZZua
BAwcyAAbAUVFRfM011/DTTz/NZ86cOR/
XS4miX79+fN111zEzbw33+Q8HLpcr1el0zgi3HAprSq2FYt26dWjTpg0OHDiAF154ATNm+K/
RwhaHC8Xu3btx880348iRI5g6dSpmzJiB3NxcXHfddViyZMkFX//v4HQ68dhjj6Fly5Z455130L9/
fyxatAj/RtK7iIgIPPTQQ7j+
+kvGqV1xnrnsssuwYcMGvPfee+jUqRNee+01dOvWDYahggUuJfLy8moS0W0jR46MGjJkSEy45VGUcp
i5t67rfNVVV3HTpk05MzOzQPs3DIO9Xu85vy38HQtFXl4eX3311XzNNdcEvUHpus4dO3bkmjVrclZW
VnFeWpj5wlooTp48yQD4qaeeCmpPS0tjKeV5X+8CoywUlwhcyELRqFGjoD9ycnIyA+D169eH4xq7YJ
R2C0U+8fHxmsvlmuJyuT4MtyyKPymVFopNmzbhxx9/xL333ouoqKiCdiEEYmNjAQDff/
89Bg0ahNatW6N8+fK44YYbsHjx4n+89hdffIFdu3Zh1KhRiIyMLGjXNA2jRo3C4cOH8dlnnxW0OxwO
LF68GBMnTkT16tXxxhtvAAC++uordOvWDbGxsWjfvj22bg39fjEMA48//
jhatGiB2NhYdOvWDSdOnAg6pqj5C8PMAACfzxfUHhMTA6I/
U5n81XqbN28GEWHv3r0YOnQoypUrh6VLl4KIkJSUFDT31KlT4XA4kJqaWiDns88+G3TM0qVLcccddy
AmJgYtW7bElClTIKUE4Pe7uPvuuxEXF4fGjRuHzK+4tGnSpAkAYN+
+fQVtf3VNrFy5Eu3bt0dsbCzq16+P3r174/vvvy/
odzgceOmll3DXXXehUqVKqFevHp566qqmQtZcvX44uXbogLi4Ol19+OR577LGC6xL48z7YvHkzbrnlF
lSoUAHt2rXDpk2bguaRUuKJJ57A1VdfjRo1aqBKlSpcccUVeOaZZ3D//fdz+/bt+Pnnn/Hoo4/y
pEBT/Pszce8qUKQyAf/755yLfBFJTU3nGjBm8aNEiXr58OTudTnY4HLx///6CY/6OhSJ/7V9//
dVyTQA8YcKEgja73c4tWrTgjh078pIlS3jv3r2ZHB1atX52bNmvGCBQs4KSmJmzRpEmKheOKJJ1g
IwQkJCbxw4UJu1aoVX3PNNUFWBav5rejYsSMLIfill17iY8eOWR7zV+tt2rSJAXCLFi145MiR/
O6777LP5+NGjRpxjx49uXGuZq374933777UFyTp06teD3zZs3sxCCe/
fuzR999BFPnjyZGjRpxjx49guZq374933777UFyTp06teD3zZs3sxCCe/
nRRx9lm83GH374IbOyUFwy8FksFAsWLGAAvGLFCmbmv7omCu6prl278pIlSzgpKYk7duzICxYsCLoG
y5Yty1OmTOHjx4/zm2+
+yQB48eLFBcds2bKFbTYbx8fH89KlS3ny5MlcpkwZHj9+fM8ex+fdB1apVOTk5mQ8ePMjdunXjKlWqc
G5ubsFxzz33HAPgsWPH8rJly/jBBx/
7772cAnJ2dbflgtCIrK4sBsMfjKWj7OwpF/
tqFt1oKExkZyYMHDw76EqtRo0bQNsirr77KAHjPnj0FbRs3bgxSKE6fPs3ly5cpcPku/
IkSPscDh49erVZ53fioyMDB48eHCB81vr1q153bp1Bf3nsl7+F2n37t2D5n700Ue5ePMjdunXjKlWqc
G5ubsFxzz33HAPgsWPH8rJly/jBBx/
7772cAnJ2dbflgtCIrK4sBsMfjKWj7OwpF/
7772cAnJ2dbflgtCIrK4sBsMfjKWj7OwpF/
tqFt1oKExkZyYMHDw76EqtRo0bQNsirr77KAHjPnj0FbRs3bgxSKE6fPs3ly5cpcPku/
IkSPscDh49erVZ53fioyMDB48eHCB81vr1q153bp1Bf3nsl7+F2n37t2D5n700Ue5ePMjdunXjKlWqc
G5ubsFxzz33HAPgsWPH8rJly/jBBx/
7772cAnJ2dbflgtCIrK4sBsMfjKWj7OwpF/
ytoi4uLCxrz5ZdfIiMjA7fddltBW/Xq1dGgQQN89dvXZ53fiujoaMybNw//+9//
fPtx6662YM2dOYM2dOsde75557gn7v0aMHzpw5g5g7Vr1wIAVq9eDSKlevGxZWr14NAPj000/
6NHw4ZB56dly5bYfE6dOoWYmLM7OOcrPoMHDy74oiUi/PTTT/jjjz/
ZuOaaa+B2u3HgAE0NeXl5qFQ5Ddy74oiUi/PTTT/jjjz/OOv/
ina5t9GQAgLS0NeXl5qF69uuWYfE6dOoWYmLM7OOcrPoMHDy74oiUi/PTTT/jjjz/OOv/
ZuOaaa+B2u3HgwAE0bdoUjz/++/
i5n09HT8+uuvQXIMHDgQuq7jwIEDkZaDFBc1+VEe+X4Ro0ePLvCjAM5+TRw9ehRCCKxYsQLdu3fHvH
nzULduXXwewQwePPgv87pUqFChQpUNx4441o2LAhZs6ciZ9++gnPPvssbr/9dnz//
HqP4E7fb3dDlcq0PtxyljVKpUNx4441o2LAhZs6cibvvvhuB4L8uK1SoEHJdpqWlAQAqVqx4zp8nX6bMzMxz
HqP4E7fb3dDlcq0PtxyljVKpUNx4441o2LAhZs6cibvvvhuB4L8uK1SoEHJdpqWlAQAqVqx4zp8nX6bMzMxz
x5PtZu0KABZs6ciR49ehRYGgzDwKuvvooaNWqwgqU6dOZ52jZcuWmD9/Po4fP4fP17wxZObG/
ycbNy4MapUqYLc3Fx06NDh2sgAa++OKLEFPu4cOHUbNmTURFRf3j9Xr06IFIFp06YVZPvs3bv3WY+/
8cYbi8wY2LZtW3z++ef8cYbtW3z++edo3bv3WY+/
+WU0b94cU6ZMURwH85TURwg033ID27duTRwH3PeFLZX590X79u0L2qyiNTZt2oTIyMhi3Rf16tV
+WU0b94cU6ZMwH85TURwg033ID27duH3PeFLZX590X79u0L2qyiNTZt2oTIyMhi3Rf16tVDlS
```

```
pVgqJMgND7XFE0brf7FgBwuVzr8/+vUJxXOJApc9WqVRwZGcl169bladOm8WeffcYul4sbNWrEuq5z
+/btuU6dOrx69WpesGABN23alGNiYrhPnz5sGAYzM3fp0oVbtmzJ33zzDTP781hERUXxsGHDeN+
+fUU6d+Wv3aBBA542bRovXbqUe/XqxXa7nRctWhTiCFbYGZHZ7/
wZGxvLw4cP55SUFD5y5Ai3b98+JMpj2rRpHB0dzcnJybxhwwaeMWMGX3vttZySknLW+c0sXbqUAfDI
kSN57dq1vGHDBn7kkUdCnFT/
ar18Z7RNmzaFrCGl5Lp163L16tW5RYsWIf1mObdu3cpCCO7Tpw9/9NFH/
OSTT3LHjh05Ly+PMzIy+LLLLuMuXbrw+vXredWqVdy3b1+ePHkys3LKvGTgs0R5uFwujoyMLLgP/
+Ka4M8//5ybNm3Kzz33HK9Zs4YXLFjAZcuW5TFjxgRdg5UrV2a3280bNmwoiLoonMF169atbLPZuG/
fvvzRRx/
xU089VWSUR+H7YPXq1QyAd+7cWdA2adIkrlKlCq9Zs4YNw+A333yTo6OjlVNmMRk2bFhHl8uVOnz4c
GtztELxd+FCqbd37drFAwcO5Bo1ajAAbt68Oa9du5aZmQ8cOMC9evXi2NhYbty4Mc+bN49Xr17NrVu
3LkhItXr1anY4HFy5cmX2+XzMzDxx4kQGwL169Qp5KBZm165dfM8993CtWrUKUm9v3779Lx+k+Wzbt
o1btWrFNpuNGzduzJs3b7ZMbPX888/
zlVdeydHR0dyqVSueOXNmUGjauSgUzH6l4oYbbuCqVaty1apVuVevXiHKz1+tdzaFgvnPCJj8L/m/
Og+rVq3ili1bclRUFLdq1YrdbndB1MeBAwf4tttu44oVK3KtWrW4T58+/N133zErheKSgc+iUBw/
fpyjoqL4jjvuKGg7yzXBUkr+4IMPCu75li1b8owZM1jX9aBr8P777+du3bpxTEwMX3nllUFhpfmsWb
OGW7VqxWXLlmW73c4PPPBA0DznqlDous4jR47kSpUqscPh4LFjx/
LYsWOVQvE3cDqdVyUkJLRwOp1bwi3LpUyp83hn5t4AFoVbDkXY6EVES8MthOKfw8xlAWT/
W+s5HA489dRTeOSRR/6tJc/
GNiJSueiLidPpfA9AusfjGRFuWS5FSmWUh0KhUChKHx6Pp78Q4q1wy3GpohQKhUKhUJQakpKStgXqg
HweblkuNZRCoVAoFOfAZ599hgEDBoRbDMV5IFAHZLHT6VwWblkuJZQPhaK0oXwoLhH+bR+KEobyoVC
UOJSFQqFQKBSlFpfLdSgxMfHpcMtxKVBaFQpWP+f952I5t4pLi6L+tuG+ztS1fJGwd+/
eelLK6xMSEjqHW5aLnVK35aG4MCQkJDwGoN/s2bObhlsWRenD5XKNB/
CUlLLf7Nmz1b64QhEGlEKhOG+4XK4UZk72eDxFF+JQKM4zLpdrFYAWhmG0Sk5OPhhueRQXJwkJCY8R
UXuPx3NHuGW5WFEKheK8kZiY2ExK+SkR1XK73eenPKtCUQQJCQmdhRArAHzldruVg6LiH+N0Or8D8L
3H4xkYblkuRkqrD4XiApCUlPRdIIJiX7hlUVzauFyuxUKIpVLK/
yplQnG+8Hg8zYho6KRJk9Sz8W+gTprivOJ2u0cy8+9Op/
P1cMuiuPRISEho63K5MgG0zMvLqzt79mx3uGVSXHJ0OnLkyAPhFkKhUARwuVzpTqfzvnDLobh0cDqd
bqfTeSIxMXFauGVRXNq4XK4cp9OpQkmLibJQKC4Ibre7AhE96XQ6m4dbFsXFjdPpvMvlcp0mou7MfG
dSUtK4cMukuOSpSkSnXC6XPdyCXEwop0zFBcXlcknDMFokJyfvDLcsiouLkSNHRum6/
jGAlgBecbvdj4VbJkXpIT4+vsxVV12VO2nSJBluWS4WlIVCcUFxu92apmlbBg0a1CTcsiguHlwu1yh
d170AKgshmihlQvFvExMTc9uRI0eOulyuCuGW5WJBWSgU/wpOp1MHEO/
xeD4MtyyKkovT6byeiDwAyjPzZI/
H4wm3TIrSS2Ji4jYp5W6PxzM83LJcDCiFQvGv4XK5Mpl5isfjeT7csihKFv/5z3/Kly1b9j0ias/
M6//444+
+q1atyg23XApFfHy8tmjRIiPcclwMqC0Pxb+G2+2OJqKRLpdrSbhlUZQcXC7XY+XKlUshoisNw+jo8
Xh6KGVCUVKoUKHCfUOHDq0VbjkuBpRCofhXcbvddQA0cDqdX4RbFkV4cTqdXV0ulxf+Ghwj3W53/
eTk5K/
DLZdCURgiqu9wONR1eQ6oLQ9FWHC5XB4AA3w+X925c+emhFsexb+Hy+VqCuAdAPWYeZbH41FhoIoSS
3x8vCM2NnaFlHLE7NmzD4RbnpKMUigUYcPpdA4hohcBTHK73a+FWx7FhWXQoEHlypQpswDAbQB2CiE
GzZo167cwi6VQKM4TSqFQhJV77rmnYmRk5G/MvM3j8XQKtzyKC4PT6XyEiCYD+ENKec/
s2bM3hVsmheJcSUxMvENKOdjj8fQPtywlGaVQKEoEAZ+KCh6Pp1m4ZVGcP4YPHz5A0zQPMx8hoolut
/vdcMukUBSX+Ph4LTY21sfMTT0ez4/
hlqekohQKRYnB6XQ+TETPSimds2fPTi7cN3z48KPJycnVwyWb4tzp0KGDrVGjRtcQ0WIAlzHzUx6PZ
3K45VIo/glOp3OQlDIlOTl5dbhlKakohUJRokhISLhMCPEdEX2TlJTUKdD2ixCiITO/5/
F4BoRbRkXROJ3ONwHUJ6JbmPldj8czONwyKRSKfwelUChKJE6ncxkR3U5EnaSUG4kIUspDubm5t8+f
P/
+ncMunCCYxMXEgM88CkGYYxsMAVicnJ3vDLZdCcb4YMWJETWbekZSUpCylRaAUCkWJxel09gGwkMh/
mUopmYhe83g894dXMkU+w4YNq2+z2VYBqMrMj3o8njfCLZNCcaFwuVynDcMYnJycvCjcspREbOEWQK
EoikBNhwKEEATgtvj4+JqLFi06HCaxFAESExNXM/
NtzOzxeDwjwi2PQnGhYeaOzJwdbjlKKkqhUJRIEhIS/
gBQ3twupawfExPjAjDx35dKAQAul+tDAN2Z+WO3262FWx6F4t+iZs2aXx45ckRd80WgFApFiUQIkSu
lTAUQR0TEzBBCQAjhANC7W7duz69YseJ0uOUsLQwZMiTG4XC8ycw9Afyg63qFOXPmZIZbLoXi3+To0
aPtAHwIoGK4ZSmJKIVCEURycnINwzBqElEWgEybzZY1dOjQUwCQnp4eV6FChWI52nm93mdsNtvb5cu
X/zm/LTU1tbHNZjtmNZfX661NRINjY2MbDBky5Aop5bWRkZE3AWjSsGHDhidOnIh0OBwNdV2/
H8BzgTH9AfweFxe3pTiypaamvkJEf8TFxU0z96WlpcXExsaeKmJcW4fDsS86OjokZbjX660QFxeXXk
w5JsXFxb0YOOcAgMzMzCuJyBsVFXW8OHP9XRnOxr59+3YsWbKkya+//rrWIwr58yZs/
+ll14qc+jQoc4rV67commaZhiGEEJonTp1mpSXl5eWmZn5vy+//PJ/QgiNiDQi0phZXH311W1+
+OGHnbquF7QRkWYYxt5777332PmSWaG4EFSvXn3zkSNHBn1xhtvZP31iNKFcspUBJGUlHQ/
gFes+ux2OxuGcUJKmQYgm5mziciIjo6unZGRsRuADsCX/1OxYsX/
i42NvU4Igezs7K+PHTu2KSIiAnXq1ElElgZh8zZx06dOi9M2fOpDGzQUTyqquuGuNwOGrpuv7bwYMHPe
np6ceZ2ahVq9bVNWvWfBCAj5k5JSXl2YMHD3532WWXkpNHpU6de+
```

```
+GHH5YZhqEDQI0aNWrm5eXlnTx5MuTB37x58ztjY2PHA0Bubu7GnTt3TsvLy9MBiMaNG3ePiYkZcOj
QoakHDx78KWAhEUIIqlq1aoM6depMNgzDu2fPnqkZGRlpmqYJKSWVLVs25sorr3wiKytr2549e1YII
YiZRVxc3GXVq1fvc/jw4Q/S09NPwF+UjwCIBg0a9CxXrtz1ubm5h/fv3z/P5/
Od1jQtomHDhiM1TauUkpLyfkpKyn4ANma2EZEtMjKymq7rObquG4XbmdlWpkyZahUqVGh16tSp73Jz
c9cBcDCzg4gcQoiyFSpUuNrr9e4F4CCigr6IiIjLfD7faSkl8tuY2WGz2WKJyK7r+nm+0oJh5hMAEk
aMGLHigi6kUPxDnE7nTTk5OTvmz5+vfClMKIVCEcKsWbMeIaIhAKIARAd+FOEnG4Bht9ujhRBS13Wv
ruvZRKQD0DVNi4iKiqorhCAikl6v94JWE7bZbCAiCCF0XdfTIiMjK2maRoE2CCEgpTx16tSp/
5UpUyY6Nja2FRFB0zQwc/
bx48eX5+bmplStWrXV8ePHbQAWAXg9MTFRbWUpSizDhg2LttlshtvtVtepCbXloQjC7XbfxcxTmXkn
MyfWrFmzVqNGjd4konKFDsv56aefBtetW3dImTJl7sxv1HV9zw8//DAuLy/Pd+2113psNlut/
D4pJQCAmRF4Cw76SU9P/zAqKuru/GOZGbm5uft///33D+rVq3c/
gLJSSuSHkBaFz+dLPXjw4LzatWv3cjgcdfPbT70O1zkAACAASURBVJ8+/
e2BAweWlS1bNrpOnTojhBBlC49LS0v7NCMj40Dt2rWHE5G9kNy5+/fvf1VK6atfv/
5wm81WpfC4jIyMLYcOHVrbsGHDIRERExVMc64pV65cE4fDUaNw++nTp7/LysraW6VKlXiz/
FlZWdvLlSvXSggBIir4SU1Nfe/o0aNfBSw45QFoQojKKSkpC/ft27epVq1ajWrXrv1E/
gMdgEhLS8s5fvz4W1WqVLkzIiKidv5DXggBXdd/2rNnz8TGjRs/
WKZMmesKrZeZkpLijImJaRwVFfVkvgKQ32+z2RYwc08iKnz+bAAqF/
EniWHmlQAmmv525QB0ZOaVRHTdokWLAKCVYRjLAfxsMY9CUSLQNM0N/
0tW13DLUtJQCoUiCCllXuDBkt23b99vmdkD/5d/
YXIqV658JxHdWbiRmSvUqVPnK13XpzJzLZw7m5j5Dgtl4ecmTZr0AVDWYkxRZF199dVNiaiuqb15ix
Yt9gGoa3oYAgCIaBszDwNgL9wuhOhy4403rktNTV1NREHKBDMvrVixYq/
U1NRFRBSkTAQoqthZs8CPmReYeaT5PDDzjjZt2ji9Xu9bRFS7cF/
t2rWHtGjR4hSA0UQUUbivWrVqkVdcccVgIoo0L8TMXK9evYYArjN1RcfFxbUHcG8Rsg/
8K6WuMET0JDMnEpGVlasyEQ0G/
NYOXddx7bXXNoVSKBQlGCI6AeDqcMtRElEKhSIIZs4JKBQRgTfLhuZ+AHOJ6EHzWCHEiLy8vA5ElFC
MJb9l5vpEVMbUvg1ADQD1iyF7JhFtJqL/
FHFIpyIebK8z820Agh7WRDQ8JiZmndfrfQfA7aYxGytWrNjL6/
W+AaD3ucpYFET0kpSyGxFFmboORkREdEtLS3uciHpZjIsC8LjVnMx8mIhqWnQdIKLxAD62GPMygBHF
URrOwvNSyhuIqNFfHZivUDgcDvWdpCjR+Hy+KZqmJf/
1kaWPC7rHqrj4IKJcAIiKiroCoW+vIKJ5VsoEgPeJ6AiAucVY7nsAmvmhx8x7Av8tVuVRInoLQFHKx
I9FKBPLAMQCuMEkwyuxsbFzUlNTZwAw1w/
5lpnvSktLexzAf4sjYxEskFLWIaIrTTJk2u32O30+3+0AJhRzzpVFKBNeIhrGzEnmDmZ+hYgGWSh3Y
GaraJO0ohYnotmBf82KmCU2m1+P2L59+/
ZzOV6hCBdz585NEUIoh0wLlEKhCCJfoRBCWD18Pcw8xKI9VdO0GVLKT02+FkXCzL8AOACgqan9GIBD
sFBm/kLuSQBGFdH9OYCrLGTYwczfwaQwMPO6ihUrjvZ6vROI6AHTsH12u/12IURvZn66ODIWwUZm/
s7K+qBpWg9d1ysz87xizrmamTuYG5k5i4h6SSnfsFA2PADuAFDJYtwOIqpqak5l5hDFI8ByAJsAPFw
MmTMDa6nvJEWJZsSIEVUAfD9kyJBq4ZalpKFuXkUQV155ZS3A7zxZGGaewcx3FrEff69hGO8AiDuXN
Zj5hBBiBYC7TO2niehLIrqtmGKPZ+b7iljrUwDNLboOEtEcInrS1L5P07T4tLS0oQCeNfWlCCE6+Xy
+Nsz8j02ezLwbwAtEFJIHg4iGCCEOM/PyYs65npmbWfmJaJrWk5mnmi0hABYCaExE/
2cx3zoiamlqTgmE+YZcCwC2EdEMZn6rGGLvz87O/
rEYxysUYSMnJycDgCMiIuKcvu9KE0qhUATx008//QD8GZUBAEQ0EUB7IrrMfDwzTwnsxzc4l/
kDb8kvMvMYi+6VAHoUR14ieoKZnbDOXLcSQF0AMSYZMgA8zswvmdozbTZbVyllO2aeY+rLBtDRMIwq
zLywODJaEdhCGA5gnkXfZE3TPtF1fTWACsWYdhOAakRkroYoAXSTUo5CqOVnA/
zhwTeY2sHM6wG0MrWdYeaTZgfVQN/
Pdrt9eHHODzNn2Wy2zlJKyyRiCkVJY968eTlSyo5er1c5D5tQCoUiCCFELgDk5OR4Af+bspSyCRG1M
B/LzB8G3l6vLcb8jzGz2SqQP1exnBuZ+WVmvsXK6S/wMBRWb91CiNEAXjK/YRNRf5/
PF8vMH1gs19tms+UQ0UorH4Niyp1ls9m6E1ESQsMt34mLi3vO5/
Otgl8ZOle+BSCJqLHFeiPg3864y9T1G4AjALpYjPkUQE0iCqqnQkT/
s7BwgJmPORyOu30+33soOoQ0BCHEoOjo6IORkZERgd9VbhxFiUfX9T2xsbFmB+pSj1IoFEFIKXMAQN
f100QUy8ytiCgkXwL8To4E/
4PqXLmPmceYIxmYeS1Coyj+infgd6a82aJvJxHtLkK2+wLWEfNDbwKA34lolYWiMSQiImKnrutrAmv
+IzRNu1vX9XEIDT37PC4ubqDX651vsc1wNvYx868AbjJ3ENGTRBQDizBQZt4IayfWDbBWxr6AtSUji
4hu9/
l8z6MYjrRENImZ+6elpa2vXr36zYG51HeSosTjcDhWM3NxfIRKBermVQRhs9lyAYCZbVLK4bDOR/
AbM+9AMbYnAvkIhgAw52vYQkTNrfb8i4KZ1xPRXvJn8zRzgIjmwdpB8wX438abmNoXENH7zLwGpi0G
IpoopfwwLy/
vMwvZi00gwqKt2QmTmX8RQvT0er0vWDlonoUUZl5RhHVnVsD59QWLvoX5OSBM7AKwx+zHwsxfoojYe
03TusPv2NqtGHJ/
BMAA0AfAdflRHm3bti3OZ1cowkVZ+LcKFYVQCoUiiOjo6FwA0DStHBG9aHHIbwDeK+JhVBSvSymbWW
yb7GTmSrCILDgLPwgh3mHmp8wdzHycmR9mZqtaJB/A/
wVgtlpsYeZHpJSfEVGQ1zYzJ8fGxk6Gv7pgU/
xDAlEhXovokDRN026XUvYFMK4Y82UT0etENNqi+z0hxCIA71n0rQbQ16L9ADO/DVMoLDP/
```

```
FIjyiLEYMzCQxKw4b2s/
MvP7hc9DvkJRrly58xGGq1BcaPrpun4+orwuKZRCoQiiT58+eQCgaVpI2Cgz/
wFgNoBHijHlcgDZFm/
d+wKRAiE+DkXBzEeI6DGrCAtmztQ0bSARvW7R9yUzfwVgpKnrt8jIyL5E9LHZD4OZ11esWDHB6/
UuI6JbzlXGs8i+hIgWSymtHvDdpJRXAnizOHMKIR6XUoY8yAMWnBcMw/
jQom87rENyvUQ0yUKJPAQgHUA9izGPENEfxQlrDTjEjoM/TLWAfIVi//
795r+RQlHiyMvL2x8REeEItxwlDaVQKILwer1NhBBWYaPHhRBJAKYUY7qNAD4DMN7UfiqQe6EoP4GQ
0tuBffrEs4Rr9jEM4zmE+kbsC7zFTzfNlwmg25kzZ95G6L7/zri4uLu8Xu+rCHVk/
Dts1TRtPDOvNjt0sj8t9WlmXlTMOR+RUj5s4SD6rcPhGCWlXGV2qGTmn+HPBlrB1J4thEg0W3aY+XQ
gB0Vb8+LMnKxp2hJm/
qg4QmuaNiyQQCva1J4HAPv37z9SnPkUinDgcDhmSinfDbccJQ2lUCgKOHXqVD1mXhyoBlm46zv49+O
LY+L7Hv630Fct+t4nop5WgwJWkJBQSSHEYGaeBovtESFEPBENsdhS8RLRGGYOefPXNK03Mz9ptj4w8
x8Oh6Oz1+sdgaITZRWHA8zcV0r5oUU452sAVkopVxbHhwTAM8w8yLxFA+A3m83WLy8vb6nF9s0xALE
WCapARP0Nw5gM05YGEb1NRHebj2fmtXa7fYJhGGvMY84GMz9kGMZ/
raJyTpw4MQsANE1TqbcVFwPRAPgvjyplKIVCUUBMTMwB+PfzCysUywCsDuSiMLO/
iKkOAXiUmWdb9E0DMKKIcT9Y5boA0J+Z7yeiKyz6RjHzFbD2CRgspXzdHFVCRMMMw+hvjl5h5gy73X
5bbm5ue6tkU3+DNE3TbieiNxHq0LhRCPEkgBDfjbPBzHOZubVFeGiq3W7v7PP53jafJ2bOICKfVe6I
gJPovVbJrph5kIUIO202W19d1z9C8cJaFxBRJSK61UKGIczsBQAppVIoFBcDY6WUA8MtRElDKRSKIC
IjI3sRkQwoFFMBHIeFw11gL96qcJcXQCL8D5BI05g5AJxFLL0LodEXIKKJzNwJFiGR8GeZPGxlOSGi
wcw8yaygBMz6tawiRIQQd0kpaxDRBwDORz6EblLKRJjKHDPzbk3TegasFlZKUlEsB5BpEYFxWtO0zn
l5ec8QUWtT3xkiSgEQUv01ELZ5DZnqbTDzOgCtKDSN+u8Oh6OzYRivwiJ8tCiY+RsAn8LC94b96c9H
16tXbwIAtGzZ0jLjqUJRkjhz5swxZrbKFFuqUQqFIohy5codyc3NPSqlZGZuBMBlcdjnRBTipMfMWU
KI/gBeR+g+/VL4nQFDTOTsT0EdEpLJzHPYX/
10qIUM7xDRe1LKdyz6HpVS3mWxBfIJEf1kFSECf9hjumEYyyz6ig0z92Hmxsz8kKkrRQjR2TCMmbBW
kopiKxGtI6L7Lfp6GYYx2Gp7goj2wiKLaeDcpgAwP8B/
hN+ca1YW0+12+x25ubkjEVosrUjYnxH0MbYoRgZ/Zs/
GAJrbbLZIAChTpkyf1NTUT891foUiHJQpU+ZhTdNWhluOkoZSKBQhSClPk5+QhFaBt9dYWPsy/
EdK+SxCIwK+IKJYqwyL8IehVkCoArIW/
rfa5y1kWMvME6SUH5t9DwJWkPIWeR52BywPsyxkeIyItkopV5u3R/4O7E+clUlEblN7jhCiM/
tThRdVFdVqvj3sr6USEg5LRMOJqNmuXbtGzp8/Hz6fr/
C4r2CRaIqZ1xDRYvgVv8LtJwKRNK3NY4iou67rrYnoiXOVOzBuKIDZFj4imwHUzW/Pj/
IIsKY4aygUYSAaQE64hShpKIVCEURGRkalcuXKFVUg7EsAv8M6G+J/2J/i+VrTmJ/
hVxpCMloy8xEAaRRa+fIHTdOehUWdC/ijGAYA+Mg8jpk/
E0JsQWip7xQhxPOrVq16a8uWLWYZ5miaNktKucbKYfFv8JoQYgOAxeYOTdMGGYbRDMBjFuMsS4Ez83
FN0x6AP1zX3PcUM+cBeG7SpEl44IEHUKj692oiamMx5f9sNtvjzBwkH/
trlawwb6cAABHdA4CKEx4aGDeYmSdabDvtYOYIFNqG0TQNAJCdnb2qYsWK06FQlGCklNMBhFgESztK
oVAUkJ6e3lDX9W2apkUCIQrFLgBzrLYfmHksM98C4E5T10ki+hDAPRZjsgH8AOAaU/tRACN8Pt/
7c+bMKXPHHXegTp066NSpE1566SXvd9991y0vL+91i+2MH4UQr5nDSgOhj49IKV9fvHgxJk+eXLj7k
7i4uNG6rq8iosvPdm7OhR07dnzaoEGDKpdffvk3TZs2LdezZ0989dVX+d2PGYaRRURWjqr7mdlubmT
mLE3TBkgp37QIAZ2jadpWAPMBYPz48ZgyZQpatWoFAJ+wRflyAAfsdvtAwzA+MFsMhBCziGi4hQyTb
TbbjuKGhwZ8X+4wKzXM/DMR/UFEQUXH8hWKb775xur8KBQlivT09FRm1sItR0lDeVQrCqhQocKvXq/
3MBE1LNzOzHvhz5641mLYG0KIsswc8jAC8DKAZ6zWIqIVAPqZ1jlNRH2Z2T1hwoTK77zzDsaNG4eHH
noIe/
fuPTNx4kTxzjvvbNm6dWvtiIiIwkNThRAPGoaxhCjYl1II8SgzvwBTmlxm3h4XF9c3LS1toZWJv7js
2LHjh9tvv/
26Zs2a2YYNGyZsNhs2btyI4cOHY9OmTQvi4uJWMfNmi6G7AFQoYqslXkr5NEz+DMy8xmazzdJ1/
fP8z9u2bVu0bds2f6uondkhFv7cHt3z8vLeIqK6pvleZmaX+dwBeNdut7/q8/l2oHhVT99n5lwA/
U3rnBBCfGzhVwIhxI8ArgIQolgpFCWNmJiY/
xLRAzgP6fgvJZRCoTDTXUp5DEBkwELxu6ZpnaWUHwIwJ2VaLITYwaZS34G+cURkZdrPzxrZz9xORPH
M/
MSSJUsaJycnIykpCfHxfjeOjh073pKZmdn1hRdeeCw5ORkjR47MnytH07R+gbf4aNN8k5j5fgBxpvV
/sdvtnb1e70wiCqm0GTgm2yLKoSh+69u371a73d545cqVIjLS/
yzv168fUlJS1lesWHGSYRhbLPw9dvjFJKsvpQFENAjA9ab2nUQ01jCMdRbzrSOiBgh9+OdpmtZJ1/
WJZssAAE/AmdOcGfULZv6vrusbUIwvTfYnwnoHwApTe6YQYgozz7QY9vsff/
zRG8BPzKy+kxQXA1EA8sItRElDbXkogoiLi0vPzs7eG/
j1mBDiFinlOITWsthCRG9bKRMAngk8DK2SHm0goo7mRmZOANCDiG6bP38+6tevj3bt2qFdu3bo1
PNa0aVMsHHoBkKYFyvXr2WtGnTZuk333zTsHHddb969erV8rV7d+/P1JSUvL/P1JSUvL/Fj/
```

```
bbLaeRDQfxShND+C3wFaHVfn3h5h5skW712az3VarVq18/
xVloVCUeIjIQ0Tdwy1HSUMpFIoQDMPIAYCsrKw7pZS3w59XogBm3iuEeJKZrepSvAN/mKJVZcr/
MfPlCH2DfhZ+5cNpGAa+/
PJLtG3bFkIIABhBRLsCWyRo164ddu7ciezsbDDzmAYNGlyVkZERnZiYiJtvvhkzZ84EgBVTpky5LSk
pqeott9yC6dP9Pn6ffPIJ9u7du1tKeRsRPXny5EkMGDAAe/
bsweTJk9GlSxeMGzcOK1euDMpoeezYMZjmL4C7I7vb5fP0TEhKuzMrKQr9+/
bB161boun5CCHE3Eb1HgXol+evt2rUrz+VybezTp0/U+PHj8cknnxTMd/z48d+///
77msuWLYsfM2YYMhg0bhs6dO+PYsWMYN27co4Eti0abNm1CZmYmOnfufuDADfSil1Mu1ZJCQk4O233868
6aabNr/22mtlc3Nz8ccffwAAHn/88aPz589veN1110W//
PLLSElJQXx8PJj5pBDiDl3XJ8gTzn7H4u+anMr8fwHwKTQf+XwBjYVIw8/
Nn6LrevVy5cp8DQJMmTUao+H7FRUC6YRiW90JpRpkXFVYYAGCz2erDFFoIv7/CCCnlQostgW3M/
C1ZZ5ncD7+vgDmiYyER7QCwBADS0tLg8/lQAZUlVKFX+yx927d7/
TsmXLCjVq1Gh+9OhRPNwMxft8nHnlAqM8Eti0abNm1CZmYmOnfufuDADfSil1Mu1ZJCQk4O233868
O+dZbbyE1NRUrV65Eo0aNwMxfnjlzpu0bb7yBLl3+3AkpPH9iGiolLIBET3Zvn17LFmyBKNHj0bXr
l2haZqjc+fOn73++usNoqL87HH5682aNWvmrbfeOhYfFSUSnZnYKIe6HRY6X0oyyUCn8NAt28tvvfXWkw
DQo4e/
2nx2dvbhuXPnVoqFHorXt6dXXuFHOrZt2xbNmzjEjn5k1a9dZdCM1PAQCnAFg6ogkhRjHzqzBtL8Gfz0
ouK6dXIuotpbwcgfLqNpsNkZGRbdLS0rYy8w1aVlwEPSUSZnYKIe6HRY6X0oyyUChCICIDABwOx9uF
25n5jBBigJTyTYQW4TpARNOKUCZOMnMOQm+
+zcw8U0q5wDzg6NGjP1SswPEJAIvgd9bLlw0AkJaW9gMVSgc+YMAAANgHYM/PP//c5dixx
+iD3g0E2my2tXr16BQ+2LVu2oH79+vnKxDoiatu8eXNs3749KJ9DofkH85UJAMhfLy8/70ubzfZw06ZNWzRr1qxAmQ
+mt06NDhtXr16mWtLGiQffu3ZGe85fLy8v70ubzfZw06ZNWzRr1qxAmQ
CA8uXL97rmmmuWy1Eh07dswSQiwHEKRMAMDChQt/
AIAXXnjgcLtzJz19ttvL8vVQpkyZw7/
nsrq3wX8P8Ns8x+KgEeYebRCPW1eB1+BcMc/QP4y54z+8ulA/
BHekgpAeWrpUoWjSh2FKFcl2aUQQEIgYhB5e4F+zBWKoolPFzc5QuZSYZnPjbalvPz3BtGP
2x5UjJPgMdvuYbY5rkg/H5KRmpSTdwmNKS8UJ1Arn0PDz47ZGE/
HEkpgpAeBrpUwoSjh2FKFcl2aUQqEIgYhE4F+zBWKolPJJMpUcZ+YMIYTTKhtiwER+nEy1J5j5FyIa
RUTLC5vI4+LiYLPZeNGiRV+npqbOJqJOhcft27cvBQBuuOGGZ/
Pb7HY7YmNj04loGREyX+AR0cXPPsetdvt37Ru3bpD/mcD/D4G+/
fvR1xcHCpWrHhrXFwcEhMTes6jh8/DtP8hWVfpGnaCpgcDwEgMjJy6NKlS499/
fXXl73yyivYuNnvs/Hb3v27DmUmZlZNS4uDvk/
+es999xzY3wn+34c2m81RqVKlumuZbQOHjwIBo2bPgu/
KXAzWz77LPPPdgb0Op3l76cmaNWsOAYCRI0eisCzp6Gz97fhGe1TAPUYtvvxjdGgQyAmAmJIF3ItKsUjvQUmRs3jqiMiMBxBPfgGWOAHuuSHsLyBeYpmIRQiAnJ+fXuLi4omq9ZuTgfMo5dtsLDzbAq4w6o5ZtRmifKnGMAHV1iRJRmLZgI
SfGsWHBgDMB3xN4u/FMcKiSHoPqQ2dw/gkgkSwarsEy2UbeEwO/lcfxNv0W3VZbFWrl8QFDf7CPi3UjMlLb9PmRyc1CI7FyP3Ns9uP5AwlsDVQhk9LDqR1dtChXyu1VR6EeEp7gpEyP1Ap2hKqFt3Ze7MeRvq/mRFbGjE3Hi1XcawEAj4L/
jCFFq59ahDkcHr9zutDgwS2sO1W1TDEiuydp0/GqKl7btzCF/
wr4dO1BB4dlyGHc4g6MyNs8xbaZ8wlLsVC1TCjPHVtiACzHW4h7KhqoVbXyzS8bm8GlADWbcBW
LT9Cnp8eXGFNdTpsjHfvgL6v00cxpAvJIW5WHG9vzLTjYyPtEUM8HC1jy0EWvcGEMlrYPzwxEd5S7Y1
vWn1ErFytxz9pyPhw6dcN7KZLgt6ApB5XA/
ZeUFfvTSJTdmOe8ofB7uxyDm6rV5o7ftrGO7r0H/Vm3a9zDcsugBL0HMlAHjy/
NO4UqfE7eiu6mhrgCfWdlJPsjtnwZT0+Bvt/ZZxz1s/
GE39UT0tb2JfL6qntqe0v4cB6ydDeXEknK2T5T+I5TnmqKJVmnPuWnEvFVWJl79eWN9UtQ7hfN3oRYGhzhE2VNh8ZiSvNn/
k/3AKnUtNQqlBA2M8/SBmhm3aFz/
gm4jytsfqJ4K1anzA1tCpMSHsCDYmsnlSSdI94FXbO2nCX9AZtuPs4C0KuYTDe93mnBAnV5hWpGD
CcAlovMBAZf9OtqCt7AbdBa/
yTRgQbH1Xbm3htsTU9fKjxnQ2mLk2WG1Z6Dh1xHhuaGFSVRgtmGS+LbXtHCHnTH1v+fR6HWzHbt
v9cqDRb7aoewKnLPw3ckTJaaIa0WCYXAaXexZ+pRLhIxw1hp8qbH9b96h8xm6kZTbcyy8+Kiff/
E84aqy+7GlPmVZ+4z5+7G+Z8ESYH+3zFcxMW+Ojm8kPhdrFfy7qPy4iPTr4TwMPUF+aRzWfW
xa71CbhUbdbLVw+3JTgEuFlqbKmT4/ZLKe+yXWgH7NywhbVJZ6D+JSfnPXt9mAKg70+5cT7iF
6t5vmJ+7/Y4VYRyjbLlHHKSFLaZ9UPKT+vwyYHMwY/
kZg+Rf4Iz+bVLLn4ZnvdSLJWgXX0FqHl8y3yvOzAQJ+cO5gUJSCn+OTAYJ1AsoNnmFSq
jQhGB6F9CKcHpaXF3Y3jUuTr93RWAc4Lld/
fOMSTAsnklDhfOV/
6YnnJhNQGOEW0cn9WQlOzwjz3WrD16mpkKKIq9LL3OWqYgR0+GyI1A4Xy9D5FG3b2eGo1jPGWrnQ
ZPr43iWP/Ti3mGhvHKUyNmt6xvpxaybCBA9TODJpI1EobgqwNY5AXOFlQBb3Z/qWELkLdNTYrqZEtakL4hy1o2/
AHRdcoWj/oZdlJdJpDcOiaRW6cbpQjEnkBMOthGzAlJzHaegWJxlNgBdXzIxYxFJe/
SZ5zBu4TyKFHtKK/
/E2b5zAp5O4nW5GYMi03EA1MtcgGh4YFGuqYaB3xnuXMf7V/d+mfsq9ZC1r9adp5R/
ZB44AV2AOsTPGUBdxvrB2s2nsGmAJ9C4WVd3z2AXcM/
/nrbBXw9vIyIrALc4zTb72rY8y4ALm3VwqG0kjWCLeBCLJQsT0xzj61xaLLlpyYuGvK1wGJFOLNzsv9
9yMcRqkeJWxiHOMR9jDBTb9aPBuzi1RKH9ldgYjAnWhHsm0zrsCPQwbLcnQbT/
dOHpT9FS2qAfezAH7JbS+8PgNuUoXm/OTuTf48Gif/
VwlbOGoEBs+HeVaAp7hMe9xWKmF17tHp2/
Qvp5vIQ+1UoMXjHG88cBe2PXBwH95yxTDELxbpU/FcqWU7TmbaA/P9zXU1umgVuzzG+mTCXrQAodJ0yD
0OhgDGDlMx+f2I/
gMWgnVPehUfXQYZY2MwsY6XGoLrkCBKKZXsxSr2JzR/
I1kS8DInJ6byyQxMeqKbhTm4h+5+TZmfnAj91+SoZWbQ/
fyXF/3Bz3IQDwg5YPExwKP0AO+qtoWA/
q0mJPpHP0Hr9BNiQOGPQFvQEe2QfcAt0e1UTd44ArPlmhNXwGl6WPbV8gi3RfqamurWRXHdhpTGIq+
hVw5Swp7BuR/tJ83m7b1B1i+w2sOU4Rr/X5tOy4GeyVkxW1YGjqjUyoVzfPBUixQ9t7h6w7y/
lzZqbMy0PMyt+2aWymSpTKkGYm+Sq6/
ZDuf7OZDo/5aqg+yxnJ2s0VtqXWMJlVUwNWTR/
xMo9XhG+OO5tA4+t0hZ8c9iYemo8O6O9s0RBl2Y8a3T8wQOX7Nx/
/2pRg0G60E0LPHLOp31EAFJ5c/7Jxp/vmo+8pwzjLfNA0E3uvmzsO4vVPcP9m+LXzr6X31oaJ69l
1aBDZuPwFvRjBSnP/z2zrVNZJSkORHpMWk/eNpEGNJ7E5AQrq/Bq+E0W+3NanqfqiFwAqrPj0zgqsl/
6k7DLvUNkJ7WKU0x53CjiJEBXs5yUm5ZqN/
xr9KH+9v1RVuhnx6LZZEAQ4jT2pk8wE2UMt/
```

```
NTiYmJzRYvXoz+/fsX9jt5BqZw4cC5OwN/
jZWx5j5N0wYBeAhAc9OYT+Pi4obAb9lpYurLIqIlANoB0JnZHHKqUJQ4hg4d2tXpdFpVLS7VhOTaVS
hmzZq1goi69u3bN94wjLcs9rp/
FUJ0klJ+jtC3zTMWeQgAf5XPzQBCIjqI6B4hxCe6rm+zCC98jZmvMCfDYn/J8wwiams6/
qQQoq2UcjQs3q6ZeU/g7fphCxktYea+RNQTQF9T+0+apo02DOMji888gpmrF45EyYf8FTu/
YOaNpi4vDp/J4QAAIABJREFUgMEA3oa/omth7gPQBsBACxH/
G1jrSVP7PiKyVAiFEB2llC1gUc0VwFRm7klEIVm8iOilgKOomQnM3IhM9UCY+cu4uLhbvV7vArIor8
7Mc4hoGACsXr36WGZm5oeJiYkjLeZXKEoMTqdzhhCiZ1JSUu2/
Prr0oJwyFVYYAGAYxgyL9M5ZAHoYhuEmopCbqQhlYqUQYrGUMqQWCBE9GRMTs9jr9a63UCY+ZuaqFs
rEMSJKQei2yRlN0+6QUvaCtTLxhxAiiZlfLuJzWzGBiNohVJk4pmnaaCnl+xafeSoz60UoE7OFEAsN
w9hu0TdaSukmolhT+0sA4pjZSpl4gZl9ZmUiEHnzKjPPsFhnmGEYFYnIqjT8DABtrZQJAK9LKRMotO
aHB/
6S8WZlYremaZ1PnTr1ipUyAeDtfGUCACIjI6uVLVs2pNqpQlHSIKItzGwVQl2qUQqFIgQhhB0ALJJQ
MRH1Cbyxh6TPLoLvhRBPGoax3uJBtCA2NvZpr9e7ysLSsIuZTxR+4AAFFpCvAFjtX/Y0DKO+1YMS/
siSx5j5rXOUG8z8ViDM9D5Te6YQItEwjPnmhz8zLwKwiYhWWcy33uFwPJ6Xl/
cVTE6dzDwWwFNEZN7CWc7M2+FP8GVmuRBijZWiBmCMlPIV8zknoukA9hPR5+YBgUiZaHPURoB3mLkX
maqeAlhJRLstlLTfIyMjO+bl5d3H/
rTqZhYwc3xh+YgIdru9bkZGxuXly5ffazFGoSgRuN3uRfDnyFEUQvlQKIJgZlGxYsXWFg9/
ENEkIsoJmOzPZa7DzNzfMIxl5gcRM6+Pi4sblJqaOhvAHaa+P5j5c7MyEeATWCgTzJygaVoeEYXUkW
DmTGYeKaV8rQg5T1u0rQWwCX7HxCCEEEOY+TUiqmLq2myz2aYB+NBimZ12u71vXl7eSpicOtlfDyUR
hepwBNq/DlgnQpQJZv5a07SphmEsM/
cR0WgAk80+GAA+llLOlVJ+bCHfB4HIDauqsauZuYXZL4b9hcAWWygT6Tab7bacnJxuzByyz8zMHwLo
ZrbsEJFx8uTJL5UyoSjpJCQk9HY6nY+EW46ShlIoFEF4vd7pmqZVCtTRKICZP7Tb7UlW4YBWMHOGpm
k9ALxlYen4Pi4urrvX6x1vYSbPArCIiO63mHMdEfU2txPRdCLabvVwDWyDDAPwCllnefRa+IjsIqJ5
RDTbYq0hzDwRQC3zZyKi+w3DWE2hpcMP2u32O3w+XxJCi21tJqI2RBQUr8nMv9hstlHMvNTiM/
0ihBhhGMYKC6VhmpQykYhqmD9TmTJl7gewhkJLpa8kovdg4U/BzF8DqGKxBfJ7wJ8ipDgcEd2l6/
rlRGTlsLsBQCtYlEM/
derU9pycnByLMQpFiUIIcRcRhTiXl3aUQqEoIDU1tTERjSYimBSKH4hoaF5e3iKEpty2RNO0nlLKiU
TUwtR1yOFw3JaWltYNFg+wwIP8QYspvwVwnbkxEI74OjN/
ZvGgBIChhmE8bRV1AuAQQkubn2DmqbBwHgUwkpmHIrTw2SFm7sfMS8zzAUjXNO02n883LuDYWZj9zF
wJhVKLB/BqmtZP1/
V3rOTTNK0/+yt6mv8W85n5GiK60jTmWGxsbI8zZ84st1A0vhBCTJRSvmv+sMy8m4iyEaoEnSKi0cyc
bDGmn5RSh0UWUQDbAMSSdcbNLjk5OSkqykNxMSCl3MDMK8MtR0lD+VAoCqhYseLuU6dO9WbmxZpWkK
Y+TQhxVyA8tP25zEP+olk9YEqIxMwZAO7Udf1qACEPMPijG6ycKX8BUN3CkrA1IiJiVG5u7kaL7QcQ
0RAADwCwcjA8BJOVgZmzhBBjpZRWlomJgSqlN5m60jVN62UYxgKEblnkALhD1/
UORDTW1JdNRHYiqm9qPw2gq5RyuoXV4jQR9TYM400iamjq+wyAtHBgPWOz2Xp4vd4kIgoK2QSwzWaz
OX0+3xcWjqWHAOwBEKQEBT6Ti5nftNiymEREe6SUIVlEAXwHIB3WSuFYIcRWInICUAqFosQze/
bsueGWoSSiLBSKIGJiYpakpaV9I4RgACCinlLKBJiiHIoiEKIYAwvFQAjR3WazlTMMw8rHYBWAbhbz
HSOiCCKqaur6NSIiIj43N/cji+gQMLOLmXvC4gEG4AAzx5gbiWislHKahaXDHQjLDIlUIKIeuq4/
h9BaFiCiPgBiKFDd1CwiQrdN8sckAuhg7hNC9AfwCBG1MnV9S0RfE9Fg8xhN0+7RdX08EZmjJ76NjI
yM13V9kcW5TQfwhYVFBUQ0nIgetxizgIgWFGEp+o2Zd8DkK1NozheZOS0mJuZyqMRWiouAxMTEgS6X
a3y45ShpKIVCEUJubm5mdnb23ri4OJJSNgDw6DkOfVcI8Y1VWCYRDdU07ZhhGKss9v3/
x8xNYcq9EHgjPwNTrgsApzRN65ybmzuLrFN+PwR/
zoa7LGT8nZkjzf4U5A+7HGfxoFxFRIcAhBSsYuY+UspRRURFjARwEEV4ghexPTMCQBMrxQDAf6WU3R
FaufM3IpoD4DGLMY9IKbuZFQNm/tlut9+ek5PjRuj2DQJ+G/
+xaE8AMMhizAZmflRKuQ6h2zBeIppbhLNnEFFRUY3tdnuIpUmhKGkwcy+c40tWaUJteSisEMx8+tSp
U52szP9FsJGZX2fmteYIEWaeDGC9z+fbSkRmH4P9KGJfnYiOIrTkOZi5s67rE4goxKIB4FkiigBg9Q
A7GdhqMPsYJAO422Kt/2fvzMOjqLI2/
p6q7s5Ctu7IEhZlEdlEQVBwQUQWFzZFQB1FhHRX46gj4jKCOqIi8CkoKqjpDouCOIiAigsqJOyMM4o
juACi7GtIdyBk7e463x+9TFJdDQkCFcj9PY/
PY27Vvfethb6n7j3n3B8R3FPjLR0NjwO4iYju0OnnZYvFsrisrOy7GIaDHpOJ6DAzv6M9EIr0qAdAG
/VylIieZ+ao6Vdm/ip0T7WOY3/
Ex8f3KCsrew5AVErQUKjsCJ3yx4joCkRH5GyRJGkEgtE32uiVktBS0Ss678Q+rbMuEUGWZVtBQUHzt
LS0P7QaBIIaxApoNi8UCINCoI9sMpkoEAgs1g4EMdhusVgeLi8vz0H0GvgCs9n8RsiY0EZ7eACoAC7
WlIc3n2qpLSeiQUTUm5n1QkrdzPwD6YeOFiEYvaGdTVhHRC2gWbJg5kMUzGeh56w4k4L5J6KMFmZea
LPZXvR6vetjOB9GwcwLJUn6p6qqG3Tu9xcA/
svM72nqlCK4l0aUARK6dxsBjNWU75MkqWd5efmdCM6gROkAoLcB3GRJklRm1tbJs1gsg3w+33xEpyQ
HET3BzM/
oRL2sgv4SEcrKyo4LY0JQ03G5XLoh6LUdYVAIopBlOc5ms7UlIksVTj9isVgGlZeXL0a0xb4agNPv9
6/
S83MAcAw6xgSCIYlRxgSAscxcB4BeDv0FCGZeXKMnkohWI3q5YDsH95K4qWJhaBZjDDO7dQb4T0OD9
QztAQBr09PTh+bn539DRFEDbAzWIpiE6j86jpEbKZhsKip0FMEU3RMQnaLbS0Q5iDYmCk0mUw9VVS/
```

```
TW5IK+T70hMaHgYiyiShHVdWvNeeXAhjg8/
leBXCNjr6nAdh1omvWMHMCRSfIQlFR0SZmjpqREghqGna7PZOIUt1u96tGa6lJCB8KQRTJycnt4uLi
TmpMMHMpMw8sLy9/
HRrDgJm3WiyWOwF8AuByneq7ATTVKfcg2mcinPxpM4C5OnU+RjDd9ecxdH6FaGPCA+BXrTEBAJIkPc
TML+v4emxg5s+gY0xwcG+Rfh6Px6WNtDgBP1ksliEAPtYZeHcw89OqquqFrz4IIDOG0bUK0Zt6+SRJ
6s/Myaqq/lOnzgYAbRAd8roAwEuqqkb5gRDRYArmCtFztHQj6AzbQVP+E4Ihp1fp1PksPz//
BwBRW7ILBDUNSZIGk05OnNqOMCgEURQXFx8ym80nPU+SpHuI6EEAPTSH8s1m84CysrL3oNlvAwjmUo
C+0VCK6EENzLwCwMzQlLz2WA6CoaGf6X31AlilZzQgOIhG+WAQ0d+YebR2eYaZtxFRll7EBgdThN+C
4BKEQ6evKEL5Lm4pKyt7k6JzdRQAeISI3qfovVReANCeKuwqWoGNAG7TuSYHM+8IBAJf6MyCrGXmNO
3yDDMvMZlMD6qqugzRacLHMHMvAHfraPgcwSiRfpryPYgRyQPgR6vVeieCe8jIOscFghoFM69gZr1c
K7UaYVAIoigvLy/Oz89fEPI70IWZH1NVtQOAv2jKi2VZ7uv3+1/
RCVUMR27oevLrrLWDmX+TZfkpIlqqHQyZ+du4uLi/IJhEKSoEE8AP0J8dWQOgr07/
rzLzLTp18iRJmqyqqp6/
QlHIp6MLgJf1rkunzjEi6iVJ0v0xvnL+GnIE1RpXLgpuiqYXcVKC6ARUAPB/
qqouYeYvdSJYNjNzutZJFcCXNpvtPp/
P9zkRtdIcy5IkqQ4F03tr+S+CRoM250YhEU0B8ISO7r0ABhw7dqw9MzMAZGVlidBRQY3G7XZPcbvdk
4zWUdMQBoUgCiKyHD169CciitR2OsTbkiR5SGdPD0mS7vH7/U9AP2QT2i/
uk1AAYLiqqh8henD9wWQy3RqaBYkKfUTQP6IhgjkxIoSSZEUl6GLmxcxsQfTSCBBMZz1Dz+CRZXkQM
ychmJSrSkiSNJCILmadbcURdJZ8WmfGYDEH98B4U69NnZkHMPMSm832FIDFRNRWc3gXgjuEao2JlVa
r9Q6v1ztfJyR3GYJGiJ7uPcw8CYCes9pjqqpG1WHmYyaTaRCAd/1+/
4qEhIRkAJBlWSx7CGo0DofDoSjKY0brqGkIp0xBFMwcd8EFF6Qw88M64X7LJUlawMwrdao+qKrqfaS
/VXW1kSTpTlVVXYgOR/zNZDL18vv9r+pN/TPzIQAJ2i/
y0IyL3kzGRiJaAX1Hy5eYebzegA1gGDNvJaIqr/0z81AEw1e/
0Lm3EwDcT0TaVNzrieg5Zl5XlT5CfGGz2YZ5PJ75AHpq+ikM5eHQGmmbrFbrAK/
XOxWapQlm3gZgCensz8HMxZIkPQIgKhU3ggnORsdYjrozEAg8D+AGIkL9+vVv3bVrFwKBQFzVL1MgM
IS7mNkMYKrRQmoSwqAQREFEDS+88MIHdb7I/2symZ72+/3f6EQ/TAfQm4ii1vBPUcNwVVWfAqBNF/
1HXFzc9T6f76+kkwAqtKRSCo0REmpT62QJBL/Up0A/
FfjckD+F3qzKS5IkfR4IBL4lzRbmsWDmJ4joawQNGK2BMjeU50HrsPjf+Pj4oSUlJTkxBmW9fv5ts9
kGFxQUvAEdPwfS3yRtu9ls7lVQUJAJ4AFNe0UA3HrGBABIkjSKmV9GdMTJRGbuoTM7AgSXyoaiwoyQ
JElJAKCqqjAoBDWaUOST2MhOgzAoBJVYuXLllVu3bpVNJpN2EN1DRPeHdrjUDmxfIhixoXXEO1UmMX
MfaJw9mflAKI/CjQBiLcd4oWNM6BHyZfgrM3+gYyCtZOZBekYIM39ks9kmeL3enBiRFnq8lZ6ePiU/
P/
8r0uzfgWBkRhyiE039HspomR0j7FaP381mc1+v1zsCgL2KdfKIqJfP5+sMICqkFMBMInpFryIRjWfm
UYgO//2QmXcQkV6W1QcB2EiTQIuZywDESZIkljwENRqXy/
WS0RpqIsKHQhCBmZPq1av3IYBKu40ycxEzD+VgRkbtksGPId+D02VMfIzge6lN/
ZxvNpt7hAbj92PoP6SNzjgRRHQPgAlaA4mZFzHzJTGMif/YbLb7PB7PXOjvE6Kna6HNZnvQ4/
FM1Fmi+QnAPgS/1itdiyRJvf1+/1+gHxmhh0eW5ZsDgUBf6C/f6FHAzD2YOZn1t6b/
HbENkw+YuQ2i81BsMJlMzwGIitFn5mnM3BMaXwtmLs7Ly/
s89P9ihkJQo8nMzLQ7HI4xRuuoaQiDQhDB6/
XOB9DUbDZXMigkSRpGRM8gOrvhrvj4+L5EVBajyQ+Z+UA1JGwioqUA/
l6xMJR8qrff74+LsbEYEExDrY1iOBGjmPl2RGfI/Fd6evpgSZIm69TZHRcX19/
j8bxUjRj0VaFkV5mITjZ1kJnfgCZSBsFr6UVEdfWSUCGYQ0OPgYFAoA0zz6mKMGYulGW5j9lsPsTMK
2IshbSIseSzCUA+ovcz+F2W5dE+n+8bbXulpaUzS0tL+5aUlAw6fvw4jh07hoKCAhw5csTzww8/
PFpUVHQkpEsYFIIajSzLw4hI1/G8NiOWPAQRmHl7gwYN4PP5KhoU40I5B/pqzj1msVhuSUxM3Jefn/
8eEWmn6z8B8AYFd8+siBfBME/tHhP5RDSAiFqqqlocHsSYuZSIblVVNQ/
Af2L4EbxPRCtCya8qaswJBALtmLk+AKiqCmZGaWnpzI0bN+Z07txZISIwM5gZqqoe+OWXX555+
+23r/rss882d+zYcZ3JZLo2dKx0586dE/fs2dOxXbt2STabLVKPmVFWVvaTxWK5NNQvmBnl5eU7f/
nll3/
OmTPnwZ07d04PRUWG+yrfuXPnu16vN7Fly5ar6tSp0z1U7tu7d+8HBQUFA1u3bv2YJEkgIqiqClVVU
Vxc/
NuhQ4dWZ2RkDJEkKSVUB0ePHv3PsWPHnmrQoMFN4f7DGg8ePLgoMTGxSVJS0lXhY4FAwJ+Xl7eisLD
QSUSS1WrdnZKSUk9V1YhGr9e7oU6dOm1kWU4L1wOAQCAQKCoqsiQmJj4UaivcplpcXJwoSdIKSZKSm
Bl+v7/
iI4m1QZgNQBaAWQBgMpmEQSGo6axSVTWWYV9rqdJGDYLag9frfebDDz980WKxqOXl5X9mBqsUACRJM
hORHPZR8Pv9pczslyTJHB44QoNRETOHRx/
ZZDIlERF8Pl8JM+uGRQrOLZj5uCRJfrPZXMdkMpllWUZZWdmusrKyPABlRPQlgktQvRVFWW60XoFAU
D3EDIWgElardUJ+fv6BVatW+fPy8poB4Pr167e/
4IIL7ggbBYcPH553+PDhzQAYgMrMTETqhRdeeENqaurNq1atKouPj38zISFhv6qqfMkllwxKTk7ude
TIkTk7d+5cK0kSBwIB9cILL+ySkpLSPi8vL+fgwYNbmJnDx9LT05snJyfX/
+2338KhkpSQkBDfunXrp+Li4tqWlpb+KMty3G+//
Tb96NGjHgAgImrUqFHbhISEhnl5eZsKCgqOSJJEGRkZlzRs2HB0IBA4+uuvv75YXFxcFj4fANq1a2c
vLCz8fvfu3T+Gy1RVJQCQJInq169/8aFDh34aFDh34gOClBs/
```

JyMi4PCUlpc2OHTs+Ky8vLwZA+fn5w6644gpPXl7e+uLi4qPMTOG+kpOT6zdt2vQBr9e7fM+ePevD7
YTPsVqtzQoKCnZVLLNYLHEXXnjh7RaL5eIdO3ZMLy0tLUToQ4CZqUmTJt337t27uqK29PT0FvXq1bu
7uLh4y65duz5hZomCmOrVq9e5pKTk8NGjR/cSEYWPhepKiYmJDUpKSjySJMl+vz9ARGQymRLq16/
f3Ww2Ny4sLPxp165drKpqfFJSUlLjxo3ry7Is7du3r9Dj8fiJCETka9iw4Z6kpKR2RBR/
5MiRleXl5cVERKqqSn6/35yYmNhGVVV/WVnZ7wAkIiKn0/nSO+
+800uSpONn6v0WCE4HTqdzRCAQsGVnZ4uw0QqIGQpBlfB4PLcCmAegv81mi8qHYLfb22RnZ//
6yCOP7CoqKsrKzs6eWPG41+u9wWq1rjwdWrxe7z+sVusL1amTn5/
fRJZl85neyVJRlAJmHuF2u3V9PTweT6rNZjta3Xa9Xm+a1WotqOr5hYWF9ZKTkw9Xt5+TaLj+kUceq
RcXF+diZisRwWq1ol27dli9ejXCy0czZ84MGzyJR48evTotLW3F6dQhEBiNw+FYC8Dndru12w7UasQ
MhaBK2Gy2LzweTzPtYOh0Oq9n5k+ZeSaAx15//XXdkM3TZUyE2qqWMQEA6enpe05X/
1UgZjbQUzEmAKA6xgQAnG5jIqRhNQDY7fbGAF4gouSCggKsW7cOshzcgoOZ94fPJ6JiAMKYEJx3ENF
6BDc4FFRARHkIqkx4MFQUJdHpdL4CAKqqXg/
gabfbLdLQhmBm1WgNZ5Ls7OxpRLQQgK9iuaqqkCTpHwbJEgjOGi6X60mXy6WXZr5WI5Y8BFXmnnvuS
alTp44FwGFmXuR2u4cYrammoSiKh5kzYy15nE+MHDlynSzL14R9azi4UVocgAJVVYdnZ2d/
YaxCgeDMYLfb75NlOSMrK+v/
jNZSkxBLHoKTkpmZOVyW5VeZeVAgENh87NixOgsXLsgsXB9PTweT6rNZjta3Xa9Xm+a1WotqOr5hYWF9ZKTkw9Xt5+TaLj+kUceq

47EPffcg5kzZ+LAgQNYsGABHn30UXz0UczAgBrDpk2bcP311+PSSy/
FhAkTYDKZsHz5ctx1113YsmUL6tSpY7REAP97nz0eD7Zv347x48dj/
vz5yMnJqdb77HQ6sXHjRmzadPqGImFQnCVGjBjRFsB/
i4qKEhYuXHhOGBMA8MILL2DPnj3YuHEj2rRpAwDo378/Jk+eDKvVarC62s2yZcvw008/
IScnBz169AAA3HbbbRg3bhzS0tIMVldzSEhIwA033AAA6NevHy655BJKkZmZi9erVkfsm+PPMmDEDvX
r1wrx58yJlDzzwAAoLT1t6kTPKnDlzUF5ejuXLlyM+Ph4AcN9996GgoKDGGBNA5fcZALp3746uXbti
2rRpeOyx0xZ5fUqIJY+zwJAhQ2Sz2Ty2tLQ0Y+HChdWfmzKIkpISSzJ8/
H3fccUfEmAgTNiZKSkrgdDpxw03IDU1Fe3atcOLL74IVT39+bjWrl0Lw74IAV30Iju1Fddddx
3WrFkDAAgEAmjQoAHuvffeSvXWrFkDIsKnn34K4H9To40aNcK6FF16Ihx56CH6/v8r9vP/+
+yAi7NixAzNmzAARVfpS//TTT9G3b1/
YbDZccsklePrpp6OGKejatpTvVjRv3hyDBw/+CoFgvz86L2uwm1W1A4gypg42fVOmjQJKwq+//
lqprZtvvhldunSJukSsJ9Jk9hmBmZ8+bG2+8oFzAeNvWrDvNxcizoUF84YyhBF16Ihx56CH6/v8r9vP/+
+wzPnDgAAPgqVOnRs4rKipis9nML730/+a
+uuvGQD37NmTFyxYwHfeeScnJyeH793U0/z+Ng6/
Qy1btqx0P+bNm8cAeOnSpczMfPjwYb7gggu4ZcuWPHmTB7Bo48ZcuWPHPmTB43bhybTCZesmQJMzMfO3aMMzIyuF+/
frxo0SLOysriXr16Rd7x8H1o1KgRf/DBB3z0o0CF20BwcHx/Pe/fujfT78ssvs8lk4r///e/
88ccfs91uZwD82WefRc6yjvOzNy3b19OT0/n1157jRcuXMj9+/
dnAOH3bO7pvJdVvN8fvfjiiwyAR44cyT/
+WZOSkriAQMG8ZxNE8+Pt0GxfZ58yNlS5YsYQC8bds2ZmZetmwFGm+CVK1dGzunUnUqNkoA4bSS6XK3J80a
JFnJSUxAUFBVXuh5m5WbNm/OCDD1bS3L9/
f7788ssrlb344b3144ouc mJjIR44ciZSZZZu2LBhJSPg8ccf/ccf55SUlMgPaYpWAvXnp24
+42vueYaBsDx8fHcv39//v333yPHq3K3K9EydOZAA8ZcqUSm1x7dZ+bHl9//
fRTpfsUNijKy8+8uUTkBpFMY9Fi=9dD4...

```
+OhIQE3HrrrQAQGRybNm0aGWTC/6j37o25502VKCwshM/nQ0pKSqXysL+Jtv3w/
Q3TrVs3NGjQAF999RUA4Ouvv0aXLl2QkZFx0r5vu+02LFq0CD/+
+COOHj0ayfFR1es1m81RP8iDBw+Gz+fD119/
DSCYT2TIkCExNXi9XgDRP+wAcPz4cfj9fmRnZ0d0EBGysrLOeFhh48aNkZubG/
GLGD16dMSPAgjeo+3bt1fSde+998Lv9+PAgQOQJAlLly7FwIEDMWfOHDRt2hTDhw8/
ab4Dq9UaOSd8b7T+BOH1+eq8e+Fnqm2rppCRkYEJEybg119/xfDhwzF79mz8/PPPp/
UdKCgoOKnjcdOmTZGTk4OmTZti7NixaNKkCZ5//
vlKPh56xMXF4cEHH8SqVauQnZ2N7777Dh988MEJ61R81Kb8Ju7btw+JiYkRQ2fYsGGYP38+XnzxReT
n52P58uVVamfGjBKYNmwY7HY7tmzZAp/
PVy0dIsrjNKMoShcAG1wu1zlvrHXq1AmJiYnIzc2NfGVoeeONN5CUlITJkydHvviMZtCgZQg9ezb27
duHr776Cr17944M8o0aNQIAfPnllxFP7jBax9wTU1Cinr/
BgcrIQWyLCgAED8NVXX2HSpEn44osvkJmZWS0Nbdq0QefOnbFz504Af+56L7roInTt2hVfffUVWrRo
gQMHDmDgwIExzw//mGm/5oD/3Zt+/
frBbrdXOhb+MjtThL3iw6iTz75JPr164fExEQAwXtUULKCN998M6pu/
fr1AQDNmjXD1KnBVZl169Zh8ODbDGDNmDN57772Y/
R45cgSdOnWK9AFE35vw33Xr1q3y9YPN0WPHjqFBwZVrne2ISIMHToU7777Lnbu3Il27dqdtnegXr1
6Vcqh0L17d3Tv3h2qqmLmzJlQFAWNGjWK6j8j8Wd911F+x2e+TfUywqPuuz/
ZvIzMjJyUHr1q0BANu3b8fChQsxefkJk3H777dVqa+rUqejejbzbty8cDgeAaIfvkPTRR/
bs2edHudOnUwZMgQLFiwAKNHj478aqk4f478gwGCGysrLOeFhh48aNkZubG/
5ORkfPTRR/jyyy8r5Ym49tprAQS/
QiqGXMnqf2mjVrEB8fFX6X+8DGGKq9ZbwWp20IHfOMz+Fe6e+eNCdFMQrkpbbyMFO/
o66u7M01rtZ0bGdGzqlaMYYFvMBkEMpXDuMAw2fW+E1OE3FIv5uLHI8OAlSIfs5b/IFIXXnRZ18b2OI
IkSz9c9xY9rbyvfP3Jtju7rKjpXAyvMqtJJQpGV9d9/lm/YXrKIm1ck3YVJ+u8gsZRrXfTQ+ZGIn9ag
...
```

```
dfefXVV88zWouB5Jzm9o7iFN7fvLy8Bn/
88ccNV1xxxSKz2WxspqhTxwgHnPUA1JOedZ6Ql5fX0GQy+axWa57RWs4yJ8yRTmdLxfmCw+G4nYjeU
VW1a3Z29g6j9QjOD5xO51JmbuxyuToaraU2oyjKNwDqu1wuEWIjiImiKP9i5ny329335GfXHoRTZjV
xu91LXC5XfWFMCE4nWVlZ/
Zm5ntPpnGG0ltrKkCFDZACXBgKByUZrEdRsmPkHSZJ+M1pHTUPMUFQDp9N5uaqq1xYUFGQtXLgwcPI
aAkH1UBQlAOAWl8v1tdFaahuKorwEYIzL5Tqzu5UJBOcpYoaiGjBzDhFdLYwJwZnC7/
e3BfDJkCFDLEZrqW0w892BQOBZo3UIaj52u31wZmbm9UbrqGkIg6KKjBo1qimAT2RZ/
pvRWgTnL7NmzdqqquojVqs132gttQlFUV4mokYzZ86cYrQWQc1HkqQxsiz/
w2gdNQ0R5VFF3nnnnZ0ARhqtQ3D+k52d7XI6nZ0VRfnd5XK1OHkNwZ9h1KhR9QKBwCAiUozWIjg3YO
Z/
EVGy0TpqGsKHogqMHDmyoSzLXwYCgaGzZs3aarQeQe3A4XCsJqJyl8vVy2gt5zMOh+NbIkpzuVytjN
YiEJzLiCWPKmAymf4uSVIjYUwIziZut/t6Zk5zOBz/
Z7SW8xWn03kvgJRAIHCH0VoE5w4Oh+N24UMRjTAoqoDf759bVlbW2WgdgtqH2+3uTEQORVEeNVrL+c
aIESOaMPMMAO/
NnDnzJ6P1CM4pHpVlWYQXaxAGRRWYNWvWd3PmzNlptA5B7cTlctkATMrMzBRJdE4jZrP5C2Ze6Xa7J
xmtRXBuQUT/IqI/
jNYhOMdQFOUtRVHE14vAcJxOp89utzczWsf5gNPpzFYU5VxNrS0Q1EjEDMXJuYWZjciNLxBUIiMjI0
6SpN9Fjoo/
h8PhGAPg5vLy8iuM1iI4N7Hb7QPtdns3o3XUNIRBcXLaud3ux40WIRCMHz9eLS8vb2i1WkuN1nKuYr
fbbyCiTAB/
mzNnzmaj9QjOTYhotCRJ04zWUdMQBsUJuOeee1LKy8vrAWCjtQgEADBnzpyDzNxeUZRjRms518jMzL
xYkqRFAOZmZWUtNlqP4JzmW2Y+4c6bAkElFEVZ5XA4co3WIRBoGTFiRHdFUfxG6zhX6NSpk1lRlN/
tdvtzRmsRCM5XxAzFickA8K7RIgQCLbNnz16lqupdoc3EBCehU6dOPwNYl52d/
bzRWgTnPg6H46YRI0ZcY7QOgUAgOG3Y7fYRDofjoE65iEwKoSjKfkVRvjRah+D8QVGUrxRFET44GsQ
MRQzsdnv9zMzM/
kbrEAhORHZ29mwAgxRFORous9vtvwJoO3LkyCeMU1YzUBTlP4FA4AOXy3WL0VoE5w/M/
AMzHzBaR01D7OURA0VRPgBwqcvlam+0FoHgZNjt9nslSXqNmb1E1JKZAWC52+3ubbQ2gyBFUXYw8w6
3293DaDECQW1AzFDEgJmTmVl4ggvOCbKzs+cFAgEiopYAQEQA0OKuu+5qYqwyY1AUpYSZHxPGhOBMk
JmZee3IkSPFdgwaxPblMXC73f2M1iAQVBWHw7GXiNIrlhFRg6SkpGsALDBIliEoinKQiMa4XK5FRms
RnJ9IkjSWiJoDaGu0lpqEmKGIgcPh+LuiKGajdQgEJyMzM/
MwgEbacmZOYOZa5TvgcDh2AngzKyvrLaO1CM5fiOhHAHlG66hpCB8KHUaOHNnZZDL9y+VyiRkcwTmB
3W5fDKA9gMYA4iUp+K3AzNsaNWrUZvz48aqR+s4GiqIcUlV1fHZ29ttGaxEIaiNihkIHk8nUGMBqo3
UIahYFBQVWozXEYsqUKY/feOON9zHzUwBWq6q6nZlVVVWte/
fuvd5ofWeSG264waQoym4AWSczJqrzDJk58U+LOwU8Hs91RvQrqDoOh+OaESNGdDFaR01DzFAIzlve
fvvtSyVJ6sfMRZIkFQEoVlW1iIiOt27duu3WrVvXASj2+XwlzFxsNptLnE5nsV5bXq83TVXVzUSUC+
Bhm812FAAKCgp6BgIBsyRJ/7JarQUV6+Tn57clIpfJZLKnpKRs0RxbTES/ArjIZrPdGy73eDx/
Z+b66enpY6p6ncxcx+v1fgugARENtlqtK0eOHHk1gDtMJtPtL7/8spKWlraiqu2FKSgouIKZ/
9BeV0FBQU+99pg5wePxfAxgfHp6+oYK19SNmTunp6e/
Vl0NXq+3u9VqXXWic5xO5zJVVT90u92zTnSex+NJBbAOwEqbzfaQ5tjTcXFxs+vUqbO/
Qtk8AJcx82KTyfR+amrqb5o6kyRJennJkiUlxcXFpsTERFN5ebnpsssua6Gqqrpp06Y8s9lsCgQCJl
VVTUQky7JsUlXVpKqqKfz/kiSZEhISki6+
+OLBkiQ1io+P7+b1eudv27ZtWbgeM5vi4uKSy8vLfQBkZjYRkQmAyel0imRdZxmHw/
EhEV3hcrkuNlpLTUIYFDo4HI6hAHxut3uJ0VoEp05WVlYfABOYuQ6ARCKqAyARQJ0qNlEEwM/
MfovFkmQymeJC0ROB8vLy/aqqliUlJTWXZVkCoJaWlm4vLi7eDyAgyzIuuOCCrrIs15EkCcePH//
v0aNHtzKzv379+l3r1KnTAghGY5SXl+/ev3//0tTU1Ix69eoNkiQJ5eXlO/
bs2fNheXl5KQCkpKQ0TEhIqOv1ereGy8I0b978zoSEhNbhvz0ez7IDBw58Gzo2NCEhoc3x48e/
3bVr17KK9dLT01vabLZrd+/
evaCsrKyk4rGkpKT0Jk2aZDJzyZ49e94vKiryAECjRo2uSUtL611SUrJ5//79y0pLSyMGWPPmzYckJ
CS0VVXVf+jQoY/y8/N3mM3mhGbNmmUCSC4tLf39wIEDOX6/
vxzB2VEpOTn5ImaWCgsL94fKZAASEUlJSUnNU1NTrywpKdmVn5+/
iYgkAJIkSXEA5NTU1MsLCwu3+Xy+snB74Xrl5eXFZWVlnoptJiUlNZMkKYGZ4ff7j5eUlBwhIslisa
TLslxHVVW1vLz8qKqqqizLCbIsJwIAM0NVVahqzV01KisrS/
3b3/4m9nc5SzgcjrEA+rrdbjGbVAFhUOgQil//
xu12K0ZrEZw6WVlZqQBeAzAUwHcAEB8fXz85OTky+Pr9/
oKCgoJfAKhpaWkdTCZTEjP7jx49utnn8x0DgMTExIZ16tRpWbFtvyunNbUAACAASURBVN9fIElSgiR
JcYFAIDLglJaWFpSVlRVYLJYGkiTFh8sDAZEh+3xFlmUwc2kgECiKi4tLNplMFkmSIEkSktL95SXl
x+WJEm22WztZVmWiQh+vz/f4/GsDgQCxfXq1biOTm5jSRJICJIkgSTyQSfz3dw165d71144YV/
sVgsDAKBQHp6OtCwc2nvJyIBlcsZObfiOg1AG/
5fD4wMwKBQKWvX52ysv379z9Xv379UZIkNQ23papq0f79+ycSUWDg2eDrUf7gfHjx//
pqys7KDNZhtWsTwQCHi2bNnygizLdPHFFz8rSZJNc+z/
```

```
AKgtW7YcI8tyRgXtKCkp+Tk+Pr4lAEvFNlVVLS8uLv4hKSmp0noyM/
tKSkp+SkxM7Ki9QYWFhWsPHDiwvkWLFg5ZliM+DWVlZbt37dr1z7S0tOb16tUbrFNvQ0JCQkuTyXRB
eOAN9VW2f//+dxo1anSfyWSyElE4DweOHj36oclkSkpJSbk1XBY+rqrqDpPJ1CxcXnFAD/
9X8W8iQiAQmBAXF/eMJEmQZbmivHgAvwKodL3MvE2SpMHM/
B8EZ0rCpANoSkSzmfke7bUC2BUXF3fN5ZdfPgNBR9uKXAcgAKDVnj17sG/fPpjN5pu8Xu/
tVqtVzKqeBRwOxzWqqppmzpwpfO0qIAwKHVwu101GaxD8efx+f1lo0DHn5+c3IaJPdU7bTkSPAKj0d
UdEh5l5O4C1RBRf1T6Z+RutMRHiGIC3AMBsrnI08m/
169fvD6CpprxORkbGswACOoYOAPwOYJRGV7EkSTd27tx5u9frXQPApqmzsEuXLm94vd7lCG6Kp6VdD
I0WAFHOaUT0IjO/oHP+l1ardZDX610GQOsgeWGHDh2GEZFe/
2DmS4koWad8batWrVoTkVVTflySpGPMPFSnzpZY/
cSCiP6mqupTMe45EG1M7I2Li7unrKzsn0Skt9Z+nJnf0NF2DMCtZWVlzwC4Tef490R0GwCYTKaK5c8
BEAbF2SFTluXeAC40WkhNQkR5aFAUxexwOF7NzMzU/
uAKzjFMJlMpABBRPBHlAKirOeUIgFxojAkAkCQpU1XVGURUncQ1q4koKnseM28B0KYa7YCZ9wH4A8C
1OoePAlBDPiFaXADu1hYS0V1Wq/VHj8ezCNED3yKbzTbK6/
UuiNFfdXlKVVW95cJNVqt1qNfrdQPorlfxBIP8ET1jAsAmSZJWEpHeR8BMZrZrC0PGRAKAVE15TB8E
InpUVdX7iKhhrHM0bZVKktS/rKzs7zG0rWLmy/XqyrI8iIgGAHhAp90VRNQr/
HfYoGDmcovFIpY8zhLM/
CMzR23KV9sRMxQamLkZET1kNpunGq1F8Ofw+XxlsiwjPj6+CSpPNwMAmHkpETl0qrqZ+TroDMwnYBU
zZ+h8Je8NDZLVCUHMC82mRA0oAPIBFBBRC51j7zNzDyLSDpSPp6enL/
V4PB8AuElzLCc9PX2w1+t1M/
OAamiMxWRm7kVElabpmflAfHz8zV6v9xEA98aoG4vfAehd7w4iep6Z9TJiTgHwV53y7UR0CBqDhpkP
ElEDvc6Z+XkAHfWMxVgQ0UgAlxHRozqH1wNoRkQpOvXuU1U1A8AknXrriKhnxYKwQVFYWLglKSlJDH
Bniezs7DcARM0u1XbEDIU+36mqethoEYI/
R4cOHcIDq54xMZuIRuiU72Pmf4cGkSrBzN8ys4+ILtEcygstl6Tq1YuBF8Dr0P86PQ5gE/
QH12XMfEF4L48KdWamp6dP9Xg8bwC4S1PnB5vNNjA/P/8FvS/
5U+B9AFzxCzqsm4huLi0tvR7AhGq2+RN0rpeZDxHRSGbO0jk2DcBwIkrU1mHmNYieHdkdy5gA8A6A4
wDuq4bmV2RZ/
p2Z39U5tpGZTdCfKn+GiA4CmKs9wMzfMfMV2nKTyVQMAKHIGcFZIjMz8yqHw9Hr5GfWLsQMhQa3270
NwDVG6xD8eX755ZddAKLC/YgoG0CsAXQqgDer2kdo+vwHRPssFCE4uGqXWU7UVgkRPcvML8dYp/
8IwP065ZsA/E5ED2ray0lPT7d7PJ6xAB7W1NkRcuS7h4ierarGE7BSkqSPVVVdqHNsCDPXIaJ/
VrPNnwBcqi1k5mNENFhVVRcRXaA57EZwCauuto4kSW9pDUVmPhBa/
oiCmZcQ0RcxfG90YeavEhISXi8pKflO+wxDjrl7Q8sZlSCibGZeGggE1uk8+59C+0ZE6fT5fK8BeDo
UUis4S8iyfDcR3QV9f6Nai3gJNSiK0trhcPzDaB2CP4/
FYikDgEDlmM3JqqpGOemFeImIxlbVCTM0Tf4eNMZEiONEVK86eonoCWZ+RvtlHeJlIrpfR8M+APMBP
Kg5tF2SpDvy8/MzAUzU1DlERDf6/f5rEfwC/
1Mw8xZmHhsIBGbrHHNKkrSlOoNyiM0AonZKZeZSWZYHA/g/
ImqjOfYRgEuJqLW2HhGNU1X1SU2xB0F/
FD2jb4MsyxOY+YNqaN4eFxc3vKSk5BPtjEfonn8FQG9Z6QsimsTMXxNRkubYTma2IdqJFgCQlJT0NA
AQkdh36CyiquomZhZ7eWgQBoUGVVXvjbHuKTjHuO++
+4oAwOfzlYRmJYYx8/16a9fMvJiZ74D+4KLHUSJ6AZrBOtTWQSKqXx2tzPwEgFExpt5fYubhOnWOE9
EkAJM1h7ySJN0MoHvouivWKZIk6WZVVRuqqlqdwTKW7kOSJN1NRO/rDIb/
J8vyQlVVlwHQziScqM0toUE0aqlIkqS/BAKBUdDMIjLzNwjm1blap8kHADyu48S6Vc/
4YOatsiwPV1V1cQzHVz3NhSaTqX95eflUIuqkOXaciN4E8JBOve+Y2RkIBL7QvjPMfCC0lBbTETQcM
ZSUlNSqoKCgU6zzBKeX7Ozs2S6X6zKjddQ0hEGhgYj2E9Fyo3UITg/M7AdgBvAEM4+NMWD/
SESt9AaXE7T7GICXdMr3n2A9PhZTAPSCzvQ+gFcA3KpnoBDRkwCiQjOJ6DZVVTNUVY1aYpBleZCqqu
VE9Hl1wmH1CIWi3hLyWWiuObbQZrM9parqp0TUqhrNbieiABFF7Z5KRJnM3JuIBmkObSKiHUR0h06d
RwAMgyb0lpmXQ8f4YObDJpOpfyAQeBfARVUVTUR3+/3+gQCickpIkvQMM4/
VqfY7gEFE9KH2HoWWdfK0PjE6/UKWZUiSZFFV9dWq6hX8Oex2+2WKovQ1WkdNQ/
hQaHC73W8hlC9AcO5DRGUA4pn5c70QUGb+DcG03Cf84dYwiojGQpNHIWRMVCmssAIfAKijF1rIzDOJ
6CJowjxDPMnMo4mo0lQ4Ed3PzF5mXq81GIgo02Kx/FxaWroBsRNzVRlZlm8PhYdqnRzXp6enD/
V4PO8jmISpSjDzPiLKg/4swzPMnAGNs2poNuhrAI/rtDcBwJWI9olaq3UcDZ1/
XJKkPoFA4LkYGmLxNIKzI9qZIjDzGACP6cx05MmyfEsgEHhD2xczlxLRz1XVYDKZ4Pf7fSaTKcqgEp
wZKJgHZDRiLEXVVsQMhYbMzMyemZmZfzFah+D0wMylCEZ5RDnahowJro4xEZqZGI7oyIM9p2BMrGXm
jdAPD/
1MkqTdCKYNrwQRTQXQVyeqZJKqqis4mKmz0vIDMz+flpb2YUlJyQro+CacAveqqtoC0f4ju4ior8fj
mQCgyv+OmLkQwXBKvVmDmQB2QRMhwswlkiTNgI4xAWAeERVBE6LKzNuYWXvfAACyLA9k5n7QmWU4AR
+bzeYlzPyh9gAR/QPBe1DpfjNzERHd5Pf7/w6dxFVEtALVMGhkWeaSkpK9KSkpR6qhW/
Dn+BnBPDaCCogZCg2yLD+FYFrc+UZrEfx5zGZzvN/
vBzNH0jEDkWRTRdr17hMRGsivQPQX5SEAVr3IjNDXZtTSAjP/
SkTvENE8na7WAJjPzFHvIDMvYmYrEWnzKCy2WCzTfD7fvwDU0xyblZ6ePt7j8ayr5vKDLkT0DyI6pK
qqVrtHluVeqqoOQvCrvTptfgAgKhkWMy8mok8ARDl1EtEbzPyiTnOfSZL0iU7EyW4iKofm/vw/
e+cdHkX19fHvmdndgCQh2QRICAiooEQEkSZiARElFAtFpamQZCmCitIRQfRFUARRQROqSBEBAZEqxZ
```

```
9U6YK0UEIJLbC7IaSQ7M6c94/
ZjcnsDQQFlzKf5+HRzN1758xms3Pm3HO+x7NWZ2hPm9dS1rpbkqR3c3NzfxdUioxn5gdJoF0hy/
KLqqo+S0Sx+jHWhKuuKZSem5t7TFVVo1nMf8jEiRPnAxDpn9zRGBEKHcy8G8Aaf9th8O+x2+3xAQEB
JYC/
e114+IuILlyLMwFgtifaUeDplZlToalW6hMSwczZAHz0ATxh+n7MnCg4z18ABjHzJMG8LUS0gzTRpP
zHt1sslm4ul2s5gEq6aUvDwsJi7Xb7XFyHcmhmnmIymeYpiuLzZcrMLZi5kieicC1rjoPAmQCwQZKk
ccw8RzCWKJrDzFuZ+RNPDkR+LjJzEsRlqB9KkrRXURQf/
YcrYDeZTC8rijKHfEW85jPzBQCvCOZ1UFW1DATbI9Baq1/
LVguIKFZVVTsEWisGN46uXbtWttlsbf1tx82GEaHQMXHixPf8bYPBvyc9PT3c5XKN8zaQUlXV20xqL
bTP/RP6OcycJSrZZK0/
xzrocms8oet0AKI+DYCmaBmpm5PlSS6cLDjXSWgOy2LBWLIkSd8wc4HSTNaUOFu5XK458M212BkaGv
qyw+EYD0DUX+SaYOYVAQEBg3JzczfpK2UkSWqrqmqmoigrRZGawvBEfUQCY/tMJtO7iqIs0z/9e/
QhnoNvL5AjpIldLRO8f0uISLQFM12SpGmKomy9liRVImrldruHEVEd3dBGSZIWM/
M0wTW9J8tyqqqqvwqW3M3MlUWfvyvwmSzL661W6zcXL168dA3zDP4lqqo2JK3Zn0h35Y7lqg6FZ2/
zjkFRFBMRsSRJd1QIkcR9Em5ZgoODLzidzm6SJH0H5EUoxgOoDrEzcbCQ7YAdACYDEIkyHQAgjHJ4t
jR8+ndIktSBmT8RVIJcBNAcmtOiV1G8KElSX0VRvtNt22RKktRSVdUxRNRId/
6UgICAGKfT+SbEEtTXyh6z2dwxJydnMREViIIwc1+LxbLJI+bkE6m5AgnM/
AJ0iW3MfEaW5dfcbvdCfdIptITKKvCtwLhgNptbulyuGYKIwRR9VMdzfJXJZOrtdrvXk69A1pV4U1X
VZ0gTNsq/3gFZlocqirJY71QR0RgAqwoRrkoGUJJ0uiWeCNcl/
XHP2AKTyTTR5XL9bjabLSaTKczpdLYKDQ01uiT/
NyRBU7U1yMdVHyVYFys2uD2ha3msvIWYOXOmPSMjwxoTE9MnMDCwPbQciAIw8xoielow/
QgzvwngJ8GT428AGorOyczbRHvnnshEawDNBHMeA9CViHz0JgC0Y+bP9DdKaCJJzQF01R13mkymx9x
ud20IZJyvFWY+FRAQUDs3N3csfOW7v2Xm/
gA2ixyoKzAHWi5DI93xi7IsP6coykQAD+nG9kL7Ei9QOcLMWbIsP6koykBB6egPABrDVzlzvyzLDTx
bFk2uwe5vmXkHERXYrmKtUdRLABaSb4nvD8zcD8BW/Zhn+
+sCxCXDfwIQNRDbwcydAKwmoohNmzbhwoULaNmyJSRJqhMSErLtGq7HwOC6YeRQGNzWZGRkHAUAk8k
UC4EzAWAhBO23mfmsJEndiGjGNToT2yFo9c3MHzJzTQicCSJqTURPFOJMdAfQT/DU/
R4RPQJfZwLM3EJV1XK4Ds4EgIsWi6VJTk5Od/
g6E8utVmt3APOvxZlg5pXQckv0zgQkSWqtKMoo+DoTRz3/
fMpQSZPhbq13Jph5NbQur3pn4gyAZxVFGX0tzoRn62uZwJm4BE1P4luBw7DabDa/
BeBXwdglaNUrImfiF4idiZNEFEtEi71RLpPJlF9e/
l+XAxtcnY4dO0bGx8dfSzXQHYHhUBjc1kiSRADAzD43PGaeC02qWa8RcFGSpPaqqk6ETuGRmTdD2zY
RsRtAeUHG//
eSJDkgUEqEJpmtABglGBsBoCV8241PJqIsZh4muKZXiChDUZQFhdh4TRDRS263u56nBDL/
efaFhoa2dTgc00WaDii8pG43gD+giU0VgJlfLkTX4gJrTb1aCubEebYE9MJRe4jIBd1N2ZP30pSIOh
FRl5ycHEyfPh179+4txNy8eYckSfqkELGwl4hogP5cAPZYLJb2ubm5QnEvItpGRD7OLIA5zNxYf5C1
5nDtPQqneUJiJpPJu6XXLSQkxBDl+w8oXrx4dSLykZq/07klHIrFixfjf//7n7/
NMLjFSE9PfyAkJET/pOtlkScJzyehUpKkl1VV/
RQ6dUVo1SGlIRazOczMAYJ98NVE9JOnkkHPSCLaqKqqqER5luc8BSIazLwGwK8AvtFPYOa+sixv9Tx
JX0sugxAieoOITPpEUAB2IopxOBx9IXYMkiCW2j4G4BsSNyN7U5KkR6HT3WDmLGaeIYreENEwSZIOC
xIg7cy8DVqTMD2tAUTDI5m+detWvPPOOxg6dKjgpXmkybL8JjP/
KHAW2zNzDHx7dBwPCAho6nK5JhLRo4I1F0MQoWHmeczcQHAeSJLUkZk/
J52iqyzLUBTFZbVafTqvGtwY3G73SRg5FD4UyaFQFAVfffUVGjVqhODgYFSvXh0DBw5Ednb2jbYPAD
Br1iwMHChSri0a/ra/KJQvXx5EhICAAFSuXBlNmzbF6tWr/
9FaKSkpuHDhztZcuXTpUmm3273ZZDJZ9GOem/JWCJ54AbRTVbWXIAcimZmzoJOY9qx3FsBZ/
VMoM+9r1arV9tDQ0AUnT57UT/veYrGMU1VVVNGxAcA+
+HYwTZIk6WsSd+0cbzKZpiiKskKUxFcYqqri0KFDyM0tWN3KzB8z805FUX7SHc+WJOk5Imqsj1p42E
2+gluA1ohrEDP7yEO7XK5Pu3bt2qhly5bvVqhQAY8//
jiGDx+O7OxsENFXRPSOfg4zT5EkaSYzL9AdzwYgbE0PTeE0A5o6KQCgTp06+Oijj9Cvn75v2N8QUWd
VVb+Bb7TqYwAlWRM7K3CtJpPp2ZycnCEQNwP7BeLP3jIAFQS5MgDQi5njiaiufsDtdicpiqIK5hjcI
KZOnbovMTHxmvr13AkUyaGIj4/H4MGDUbt2bcycORPqO/+bhV7G/
RogWWL1+OYcOGoUaNGoiJicH7779/TWscPnwY5cuX/
8fOyO1CUFBQKhEN8+aa5sst3uQpVfQRMGLm95i5CYAWuqE0aP0+fL7MWdOmWA/
fRMFUk8nUuVmzZiKthF9DQ0PfzMnJ+UVw8zgMYIbAvjRZlgerqjpdsN6C0NDQfm63exldm4Q4Zs2ah
Xr16sHpLPCw9SO0CowV+kgHEb2iKEopZp4isGMzCul/
wczdmHmM4Mn7x6eeeqrO8uXL2zz88MNISEhA69atsW73UuhKR4loHD
OLtpY+lWV5DbTIQB4BAQF48803Ubeuz6/
Wu16sqqqD4auM+iMRbYI4UtTc7Xa3gaALrSevo4Hg+BbW2tfry1C9Oh11oSXg6tl9/
Pjx74jIkAD4D+nQoUNwfHx8vL/tuNm46ofQbrdj6tSp+PDDD/
HBB9oDScuWLdG9e3eULOnTDPCm41ayv0yZMmjUSIuCdujQAaGhoRg8eDBat26Nz6NmjVF7RwMrkRoaOgX
s2fPfgd/
3+R+M5lMXd1u9wbBy7+SJOkuZvYpL4SW3NhLdA4imgdfeecsSZLaKoqSEBAQoE+S22m1Wls7HI7J5C
```

```
us5SCij5hZL8oEInpXVdUJgq2MDVartZPT6Zwjcnj+ARvNZnMvl8u1GoC+tLUnEZ1k5vWCeeuhbRGJ
/
qjaE9H7gvU2bNq0adaBAwcWDhgwIC9K0LRpU7zyyitjIiMj25JO74KZt5lMpk5ut3uRwHn6FkAs6fQ
kmHmexWIZ7XK5NsFXu+JKjGTmVvpoFTNvJqKvmHm5oDiqpSRJVZjZp3EcgHVEdI/
eBmY+RERrAfQXzPmFiLIh2FryzA0oWbJkqN1uN4St/
kOKFSsWSkQJACb625abiatGKLxPdi6Xq8DxkJCQAlLGiqLg/
fffR61atRAaGoqWLVsiNTW1wByLxYLvvvvsOHTt2RJkyZRAdHe0TJVBVFUOGDEH16tVRtmxZvPvuu7h
8+XKB1xw6dAjNmjVDVFQUSpcujSZNmmDRokX/yv6YmBj07NkTNpsN9957L6xWK9q3b1/
g3Hv27EGnTp1Qt25dBAcHo0GDBpg3b16BdWNiYtC7d28MHz4clStXRqlSpfDOO+8gJydHaN+V6N69O
2RZxuTJmvBgdnY2unbtioYNG6JkyZJ48MEH8dFHH+VleHfr1g2VK2vfsa+++iCL/
+qmnoLFiwAK1atcL9998Pq9WK559/
Hjt37rxmm241Ll68Lu68uAcAVFVdERoa2tzlck2Hb+j6JyLawWIZ58+YOa6Q5RdB50wAgCRJ7Txlg/
rhpItFkvMAw88sPP9999v++mnn6J27dqoXLkyBg0ahLS0tP7Mck5vuTK/
NBhQoVSg0ZoqUfKIqCgQMHXKGPxb6HO3atWtx/vz5AicrU6/
fr18d13f/
sq9evXx1tvvQUAqFKqWK9auXfuKy+X6IT09/IT1EXVZWv6IT09vdrgwYPx5JNPOlKlSqhbt25KfHz8Y8iVVVVV3iTWNu2
bYvBgwcjMTFxX+3atWvHx8eX69y5s975fVNRlI6Vk1eu7rXdc00HmblP5cqXH3bt3X1WtK9ZvIiMjz
korweG5/slrUyZMubIyMiz7dq1e1x3vT+WLl2666JFi0qNHDkSVatWxaRJk3Dvvfe6IiMjldzc3J/
hiTJs2rQJVqsVy5Yty3uxWVtpWXhhyQZtFhlXtyZEbEm3S2tvqWGZkfmpP+tqVxCRWBSAMx6sZCbl
rUSDz30EPr16we32w0i6ixJUq7IIQSwE0AJ+CqZOqCpfoqiV3M/
BuAAYIxAAAR3V+pUiVRLxODG4gsy2kAMlEE6YU7ias6FOHh4XjmmWcwWmjYZQIj+Q/rBqf8a4lsGc
8+XKB1xw6dAjNmjVDVFQUSpcujSZNmmDRokX/yv6YmBj07NkTNpsN9957L6xWK9q3b1/
g3Hv27EGnTp1Qt25dBAcHo0GDBpg3b16BdWNiYtC7d28MHz4clStXRqlSpfDOO+8gJydHaN+V6N69O
nn3yCRx55BAkJCTh37hyaNm2qlzxGz549Ub58tg2tW7dG165DxwGCFSEYN32VoQz4WoWN7ZgGe9j
DGee+45fPvttwCAhQsXFlijd+/+/eOHHiBD766CNMmDABZCuWzbtx/
lP7AWDKlCm4dOksVq1ahlz52L16tUYMODvv+WoqCjUrl0b/fr1w8yZM/
Hggw+iQ4cOSE5OLrDzpNsH37dixatAiJiYmMGMlwACEcvvnGJzJ6VUqWLInatWvj2LFjAIAiatAvJ6vxdj/
iI+Px48//PhVw8mPbd9x+Vm3ov/HhJSzZQH8f/Bh8c4XlUeDVp2Ln0b5u2o/A0jB8/
eIfaPzVKpxtorHQ6b8q6iEu3yYXaP39MsgBm6FzR5rbf+8DCIz2G6zMntPGANzEX9hf8OG/i+S0Y
Oh2CrOR9tKaskBd8tcJQoI9WCzFCbtbNzw3FfSzi2nIsGYIWhVmBqHpVpyT5QsP/
ViVvbjzdeVqcsn3sjgGPBNMEqg3y+dCSpeQNW9mKGwKVW+zwqDJk8lK7gFWl3WpxH8pvjNIwNKJSv
jCgNaPlvtOd1HZV6EBhaJVGmiB0abqNwYbXU0TQpmo9cMsIw/3VVZEHDVxu5u6m/ip4Ex/
SBZbw86W88p70B1B1ELAGXv8WBOQZgkmCKmBcp3u5N8ISERUHhiNDxtq5bonj0kP7ZyR9ACQ4AQjb
1KXm3xEi7WwI/L4qtXSO8uOkO0rqbs3vI1uQqH1VqrOy6yOHmV5lExt/gbHuWbR/
ZfzbLdnYZQaQoI1RN5s7WJ1nAFZIhjeW67LfJQhlz5qEKbDbrk+15yzV2fjzk3+VORcd8u6eqaTqB
xqf4ZiXD4f/C5tW3KlSmlmQgcZVvgo3Rhv8tw+V3Rf2Avw/MmW3S3d/
CJNfDD/i/N6yP+j3KcNcXhbT0qNmmH5BrkLUMTzRjQZNtiEbY4H8/FhU0UI//wbYRk0/
1xDO+d0f0q4udeEaKz/3N6ZBGXrCa1S5w3VQuHxTnZq9BBvgeSNM1ubLHP+SI08A0r1hwzVc05wBHGLUK
yZZSC3m7ZBUNBFPfyEnfbLRYEcl2xp2H/lhqeZGpnkmcpXTxeJAN4wD8AhUX5/0WWqPsdBCbubgOiS
fGs1Wn/
+yDiT0YPfC/+ppnjDmw3Hc1A6Q1NWJngGrW+gnpxtnvMrQyHVVOhiyHXj/
JS5oQNmbmY79/XR+H0mXXJYJYXGhp8ogebJ57v/Qcaz4XLxTNHYRMQHUOEpmkHuI3QIATw8Zc5OEB
```

```
RqqqBvXr1ymtuFhERgfLlyyvz589PgqblAQB48cUXUaJECciy3Nrlcj39+uuvtw0JCcGqVZo8w+nTp
7Fr1y7ExMTk6iNPzJyVm5s7Jjk5ObpBgwb67rQ2AO9u27btHgB5uU4AZu3fv78vM5fYvXu3XkL8MrT
tDJ/EUtYSgN8hXwl2SJL0DoBx+lyQwvB+Z23evHl/
UV5v8O+pUqVKCQBfdu7c+Vok2297rkmHombNmkhMTERycjIeeuihvAoE703p9ddf935pgIiwf/
9+pKSkFHn9tLQ0ANqX0pUYPXo0Bg0ahA0bNqBWrVpo1KhRkfIBCrO/
MLxJm94SzPHjx6NTp06Ii4vDgQMHfPIybgSnTp3CvfdqCeY7duzAU089hejoaKxatQrMnP+L7Yp06t
QJs2bNwkcffQS73Z73BXu7ExAQECA6LknSK4qifCVQLzzHzOMh2NNm5kueBLmKuuNbmflTZvYp59y2
bdtMAGjTps138E3G+9UjvvS87njKo48HD9+HE2bNoUkSW0cDkdxAOjRowesVm
vev4MHD+Lo0aOX9ImMV+LgwYM/A0BUVFSesJa34kOW5aYul2sMgBhA+3vw/
D3vlCQpsVy5cpXDw32+T/
9UVXViWlraI1lZWbhw4QK2bdvm3f6MA9CRdOqUzLyZmee98MILPb744gvs3LkT0dHRGDFiBAD0OHDg
QJboeo8cOSKXLFmygNMXHh400tqClwTwtSzLaN68eV7V0+LFi50BAQEcExPj07uGiEZkZWV94HK5EB
JSIJf2M9J6pTTyftVqFHDa0f7mTNn7gY0ZyU/kiSNYGafklfPeo0hVsLsqapqf+jUPa/
EpUuXRgOAy+UkXJGmRjJ3+I5i5D7QOtndUr6ur8Y8sWLCA4OzmHmQazTf/
d5Igx6HwwFAEg4ciIEDB+LEiRPo0KEDWrVqhdEjaqV69+vESJEgHMXKDvCDOfvXz5cq8//vjjJ/
33xu7du48TkblFixXYLsr0zw3TXO+Lxxx9vB+3GnIeqqIq1alTB2vWrEF2djaqVN69R/
cff9x7vu0AogVljIXxO4Bbdu23bRu3DrmFmYwLSSHgEcV7V69/vEIEgHMXKDvCDOfvXz5cq8//vjjJ/
33xu7du48TkblFixXYLsr0zw3TXO+Lxxx9vB+3GnIeqqrxt27Yd0GlNNGrUCIGBgfj222+PbtmypY7+
96GqKpcpU6ayVxCsfv362LBhA6AJjG0G8Ln3tV69ioSEhOoXUipbgKAGMF2xloi2sLMXwjmTC
eiigCKnAfBWhM6CgsLky5cuACz2SxKHja4AQQEBPyfoiiif+tuOm42rOhQ///
wzWrVqhR49eqBVq1aQZRkVq7Z7EjBkzMHW71113oW/fvhg+fHe+65B3/+
+SemT5+OFStWQBAWFI2m9GjRw9MmDABLVq0GjZGYmIhVq1ahWjVty/
X8+fN4+umn0bx5ABmsxmJiYmoU6eOeoMBpSFPu9rFq1Cu+//
z6eeeYHD16FLNnz8aIESPy8hYPy8ilDq1auHxYsXo0WLFrhw4QJGjRqFkJAtAQ/PHHH1BVtUjRkStx7tw5/
Pbbbzh58iQWLVqE+fPnY8iQIXlPZTVr1sSyZcw/
fffo2TJkpg8eMyTIyMzOxb98G98OJl0OhIaGeEJgeGHmzs8k4hIh+JbiJ60SPRrZ+zy
Wq1drLb7cO+//77AiE7Zr585syZ43fffbfU+i5c+eAcjTwWjRogU+/
PBDpKenqxERESMCArCSiiLvuugu9gu9evXCZ599hmevX/hrJlyyI8PHz/4cOHH5g/
f37xefPmISws7IrvS3Z3744qBv3744deoeuUIiKiYqGRlZWX96GKjY5tZm5BRG96d3OiImJweTJk5GRkGKjY
3dxszv6l6fJctyh6VLL86Mj48Pio2NRYWLSDLMhYuXJiydu1ARNH5r1dV1Q01a9Zs1yc
XGnOnDnId7o7FEV5qFWrVl0AFNjOsFgsqFat2l2l/
z58+vk52djZiYmPzDa4moYYIEcK5kfGqU9577z3i3ExMTg0uKCy1aty2FhYX95X8ui5R7YeZ9R
JTIzL0hg5lXAThGYuXRwthFRMWZ+d1atWo1X7FiBQDooDoxoyEGN5CsrKxAABn+tuOgpn5p59+4gYNC
ZMmW4TJky3Lp1a547d65P5P5cCoUa04atWqHBQUxHVq1EvD/
yyQIWE2WzmESNG3NEPyAcwcvXH+VP3H+/
TpU6DKY+vWrRwXF8dRFmMTpOYT27N58/
f15UzMBFtb9p06bcuGuN0N0dGRFj7t27nGRmL58uWZFCzO
1bl7Ozs/
PWadq0aYG1n2RDcKlhb3BdWAHOIyKOG1D0VaJgMLyW+MEDWhMBvWknYHLVkQJQAXoLhdMUQjmQgBZQ
bNz4MGD3KRJE1y6Ib0cACYZzIw3Is+y+EuU+pIJGF4AwfE44wyoZgPvh3Z2ueGJiHuCmiTpkLmFMv
lSaV4lzIZYK3EeEBu8L3ce3E+leuCeSWZ5L1K7P55bf6lJ8pMLi0qXQc/BUG6KeFvYwWzM9I4KVY4u
J8g+WKcqzeoSQ6d9iLJjwWUkVEh9FFBmZZvXFs5JzA5Xk2xHfIeTmeYF8OIxKnLq8DQvcq4z8
mqMrPf5JXG0Fz6EGw2IEjGXzVELVcfQl5Iw+HRVdgTBAEQXs8J8uJ1jTqmM2p7UJYvp1G3q1iGOb
GJ+gxCu4JF5Wt1ZxN5cu5EdvACzw6d76hmPA4/gM8Ia3GsJ3EMNb6Zy7l1sIWeH9XkzRE4nQr
IBRWwmxmuFGeJgxKpKxP5ohpGpoCuMFv/6GQzE1Uyi8wMdTwe5uXxiHH/sgmM5SqhFq49fNqgybr
KFmzYg3tOFuKHqEoiTTy8GN4qdh6H1F67wWy3+kt4P2GtOkhR1j6gggWhdHeyGYymHhuR9L/P
Bt7gtwU0a+qfRaCoEZn5xhmHn8EiNlrVYH8q3WuPj7oYKbrpf+wULhDpX3P+HeXMGK7u/X4dRLDk
p99YFBIqL1UZCCeUqJ2wBgbFxBZ+E8UNuiaWL+ZbN1ktUpXDpD3QvQ5BY5QB9YNL/
Krp8NIcnI2wrHDZhqS6Aw/w8Ml52q/
XP79oFiFAVd+DPG0dY/JHmiHr7/
8+4Y3ZQD+e5JnKk0FZK+BP9mbGDw+DKVfWlNwWjTBZXqTKh/
gdkzUbAXxEvmkHOvlFsHLEh3kX+DzYpRF3LGtbi7/Ecnq4UgD4CfFiUDNtjvK42DHdZXQ7wHHD
JSI78D1GmUxQP5AASfW6Xrzu1m3WNQoOm6QV/nuDXbPt7GcJxW6ZQNw9/
```

```
nHNmzYwABYkiT3sWPHRO9rC7vdvtnhcPD06dO5Xr16XLp0aQ4LC3MHBAQsGjFixG/6OW++
+eaBqKio9KCgIM5/vXa7/bTdbo82mUwFrjffv8GTJ0++DIAbNmyY//
huu92+XP8+2e32sz169BhWo0YNJTAwkB955BH+4osvvFUfLZKSkt7q2LEjR0VFcUhICD/
11FM8a9YsttvtW+12+1TBZ+KMw+FoJqomstvte+12u60o1Rz5/
iU7HI7Z3p9PnjzJCQkJvH79elG+hsENwGazDbfZbCf8bcfNxlXjo7fzzSY/
3qcWr9jNnQZdQ6z8ViIhIeF7AB2fffbZN0qWLDlN8JJNkiQNVVV1pX6AmbMEe+CAph0gwVce20FEtV
RVvZ+IlgvWGwugnWB7ZLNHAbOa7vhBs9ncwOVyJRJRK8F6W6FVnOgVNwuFiJ5i5uHw7ej5LREtY2Yf
hTgieoGZYyHuS9GTiFKYeaHuuL1hw4aplSpVqjp1qo/ER0fWokGtdccdRFSLteRN/XbQbmjaH/
r+JpdNJtOTiqKMhUDSGlrk6W1BVc5x1rYM9C3PszxNuBLgmxjZg4jOsCB/BMARAD/
o7WYtEvOSoihTiChKN5YK7f37UbCeENYSg0cCyFPizMrKwpIlS1CzZs3EunXr+rSzN7j+2Gy2TwC0S
UxMvCap+9udW6LbqIHBP4WZCQCKFy/
+pWD4pNlsjlVVdaZorsiZYK0U8jh0zoTnRtSEmYMBzBcsNxtAU8G+
+lFojr3emUiVJOk5t9v9ociZgNaE6/
y1OBOstQfvBV9n4hcimqOq6hzBtG7M3AwCZ4KIPpck6Q9PO+3857k8ZcqUVbt3764aF+cjMjoYWst4
0Y28CWudO/U35TPM/DMEvTEkSXpVUZQ+EDsT/aHJfuudCSeA5QKHBgC6qKo6CjpngojGMPMu/
bV6cDDzN3q7PbyhqupnAmciU5KkeCKaIJjjbVXuAxH1Z+ZBumMAgB07dojlgg2uO263e4SiKKLW83c
0RlawwW2N2Wwu5tHy0PeEyAQQk5ubO46IilSix8wrJUmawVq30gJ49BXszPyHvtqDmdcQUTmPAmR+H
NAUERvpXp9JRE2Y+RXk01jIx2FoyYI+jceuYHtfInoaQBvd0A5mHsHMKwW6B58ACAMgeur9kYgmqKr
6R349h9OnT6NVq1YXDx8+/MrIkSPRoEGB+/xEIjrLgj4XnvevAoACN1hmziCisQB8EuCYuS80R+Jl/
RgRfcDMTwLQ5zJcJqIZEPRmYWYbEXUHUFl3/
CdmnkxEG6FLRvWs94HIPgC9SGu7Xl0w1llV1ZFEJMpHOU1EZQXX9DYzv6v/fEmStBfAg5IkGUmZ/
xGqqrpMJpORs6LDcCg8jBw50t8mGFxnmNmyePHiRmfOnClQNgpAlWX5RVVVmwFoUsh0/
VoHJEnqx8xrBWNxkiRt8ITQ9boWB6DlxjwhWHYTBB0kZVl+XlGU+4lolH6MmU9LkvQdi/
uOFEYigED4PuGfkGW5m6Ioyzzj+c8zl4j2QWuMpme9xWJ5Kzc3dz10CZpBQUGrunXr9kz9+vVx//
0F8k6XM/N8AKKtoDgiusjM6/
RjnifyTwU7crOgRRo+E9j3NTNHQvDeEtFXAPoK5gwmoobwjd7skGW5n6Iov0PQ+IyI+jLzEH00i5m/
IKKHCrHhddZakesdTEDrhltfcHwUMzcFoE8OdqSmpr4KYI+qqoZD8R9hNpv/
D1rUztjyyIfhUHioUUOkMWNwK+N0Or83mUxWAAUUUJm5PxGlQ/xU6QMzn/E0/
FoA326VI61W60yn07kGuhbXzHyWiP4E8Ipg2WUQ3GyYOR5aXoRoX/
0CEX3CzF8VxW4Py6G1Xh+vO+4kotcURfkRWhQiP2uJaBqAJbrjYOZ9ZrP5pdzc3IXQ3dyY+WBQUNAz
b7zxhn7ONkmSBrG4S+kIAJs9eST6sUHMPFD/
RA7gNyKaD8HWEjNPIaKDAETvUSIzdxecZwIAFUB73fEjsiy/
6na7fxFFDKDlMXTTO5EAfpEk6bgnZ6YARDRMVdWapBP38jAbOp0OzzX9BCCHiGL0Y5IkdS1VqtQgz+
sMh+K/
4y4iMjQodBgOhcFtSXp6erjL5crzEr03EWaeZrVaE5xO559FWYeZs0wm00tutzuBiCrpxuaGhYUNtN
vt8/
VPlazJLi8BECtYdj086pM6RkiStE1RlHX6mx4zZ0iS5G1hXhTTAa20cA6LG591YuavoVP9BLCHiD5S
VXWxwIZUAE1dLtcEItLnLFwEcLfgPEctFstrLpdrqSAnZRYzfwvtqVyvLvcZgJeJqJzOhkPQ2oovEL
wPPxLRAuh0KDzz5gJ4XnCeHyRJWquq6lzd8TQALymK8h0RPSBYbyqAekT0oG5oJzNPB+DjEDLzFGY+
Q0TDBGNjAcQLrulPz3X5qLACOKyq6lxvWXuFChUaw2in/
Z9ARANyc3Nvy0T2f4ORlGlwWxIcHHwBQN2cnJwz+aIT66xW65tOp3MRfNtICyGiNm63uz/
pBIyYeYvVan3N4XB8U0jS5A+kSUDr2cvMoj31mUQ00dMeXL/
9cBlAF2b+P8EN0ZsPouc4gM9FzoTHrkHw1c84yczdmHmOIA8knYieAWAjoraC85Uk366sDgAvuFyuy
fB1XP5nNpt7Q0uO1CcsTgMQDd/
2705ZlnsS0QzBuRZLkjSKBfLnzLwSQHVBdc0KWZbHKooyXT+HiF6Alnjrs/
3AzCu0l9AzuqGTkiT1ByBy4JbKsjybiBIE6w0nolaC3+0FZu4PYJJgvb3wRIi8TkhISMgrTqfzNcFr
Da4/
LiIy+9uIm42iRCiK3BfgduC9996bazKZ1o0aNUpUFWBwCxEWFpaekJCwXpbllkTUXpbldU6n8xsIVD
JFMHNXSZIaA9DrlR+zWCwtHA7HACLyqTzw5Aq8KljyODOX0ieIAlhrsVj65OTk/
E4etUYdrxPRIADlBWNOItL3CEmXJGmIqqrfCp54BzPziwAe0x2/
SESvAZgOQR8JWZZfYuZaAIrUIpuZs00mU4yiKIOguykzc1KxYsXa5eTk/
ES+8ucriCgH4tyDOEVRvtYnMjLzakmSeqmquk7gCG3xvP5+3fGtsiy/
6Xa7f9M7J8z8ChF1grhMdg+ArUT0vm5OhizLb6iqOlVgwzaTyTRcURSf/
BEA7xNRcwAV9AMex28CdPkt0BJpo7y/W+9/XS6Xw2Kx/
Co4h8F1hpknm83m+wDUueqL7yCu6lAQ0R3V/MRms5Vn5qp32nXfrjAzKYpyKTQ0dIHdbh/
uuWkWhc+06fyebr10s9kc43K5WhPRUMG8jQCeFVRM2JnZLHAYDpvN5ldzc3OXE5FPgpckSW0URekG3
6d1ALDDN/
8BkiS9zcxjBTe27wDcQ0QtBWu182yB3KsfYK3c1AXxk7IIFcBLbre7ORHpcwIuEFGTnJycb+Fb6vmX
JEnrC0k47aaqal/
9e8TMm81m8ytut3stdA4XMx8irfmaPrqUVKxYsRcvX768UL+lAmAggAdZ08KAbt5ZSZImAvB52CCiV
xVFGUVE+m2fE5Ik9fM4EyG6OUM9+RSiJMyezNwPQIHeLKwpbprz5214HYrk5OTZTz311BnBWgbXGUm
SAtloDOaDkUOhQ1XVIQDO+9sOg+sDad+2WU6n8w1mHlKUOcw8j4h+JyKfvXhZllu73e6HIWhTzsz7o
```

```
d2wg3THs4kog4j0T6EXZFmO8WwJ1BSsF6eqamNBaB3QtA/
MggjEAGb+EDpHg5lXSJKUzMzD9BOI6DXWtA30+QAAMNBisfyVm5u7mYhkwbiIXgBCSCcl7dm6ed4Tb
dE//
Scz8ziIcwC+BdCBiB7VrbeNiJq63e7vATykGztLREnwjXQcA9A4JyfnU9KaleVnIjPbiUjf6t4rcva
xqqqjBe95N499eg37CwAGqaq6SPCZ+IKIVBJrYSR4rqeBbs4ZItoNXfSLiE4CKJ+Tk5MrWMvgBpCVl
fWqLMvK1V95Z2E4FDomTZr0k79tMLiuyGazmQtJTPSBmTcDGMtaa/ECY0QUqyiKiYhE/
RjOAAjS79N75p2HIKTNzC3dbvcgImohmNMbgBVAd4GZF5nZRURW3fGvmPkNwVPyX5IkLWNxU6oBzPw
ygMcFYxNNJtMkl8u1TbBNUxifybK81e12/64fkGW5IzM3Zl2TMGjO0QAA0/RzWNPwqArfsts9siw/
qyjKh/
qIC2uiUOsA6HM9LkiS9Iyqqu0AdNDNWSXL8gpVVeeJLspTpTJcEHkaTUSlmVlUnTEKQII+UgRguSRJ
6zxbY3rWEdEBfYUIa8JfS/
SRE2Y+K8tyQ2hKnXrbDG4QsiybVVUtBsCo9MiHkZSpw2azjYuPjy+sfbXBLYbFYrEGBgaKqg9EJEuS
1B3AfEFFwmhPhYGoVDEDAAThczBzCgTVD5IktYWmnNlZMOcjVVVPQyzmlAngBPmWKi5h5sf0FQmep9
rPC3EmxjNzVQA+Dg2ApaGhoe+5XK7lEDhDhTCHmb9yu92LBTfe/sxcUr+dwcxZAF4HME6QaJkMLX/
gKd2cFIvF0kRV1XZE9LbeCCJaLEgcdZhMpicURXkEvu9rMhF9oSjKjEKu6/+YuSc0By+/
HQuYeRtrUubQjX0JYLhg2ymJmT9UVdUnERRalGa83pnwXNNPgm2Y8yaT6cWQkJCjzKzCcCj+MywWy+
/FihUT/U3d0RgOhQ4iiiAin/
Czwa1JcHBwdYvFUpTPubcV+UxBlGERM89nZlHpIwBcIl95ZwA4L3IyAAxQFCWkkByMryVJWk1EIhls
ENEu6ML7AP4CEEo6GW7WFDf7qqoqSjBeyMy5pCk5QjdvW2ho6KtOp3Omfs0rsJaIunlu5gWcHSKaBG
AfM0/
WT5Jl+XVoN159BUY6gCzybQ9+0WKxPJubm1sbgF5bAwB+hk7LgZmzZVmOcbvdYaTT92DmS5Ik9YVWZ
eNzQ/ZEtp6Cr+bGNiL6HOKKjkVE9JbAQXLKstyBiGYKohZOAG9DEyHzWQ9AR/1BImrrdrv/
z263TyUiBhAgmGtwYygO4LS/
jbjZMLY8dKiqOpo1CWCD2wCHw7G3dOnSD+Eq1UpE1NqTVBetG9rpCV37lHN6sIucCU/
ehE+1BDNPJaJ9RPSzYK3vmfkb1uS7RWb+DuBJ3XrnPAnE+qoNQLtBfaLfv4em+7ALwDDBnGSz2RzjD
DpHABAlb/
rAzPsAvMjM04iohm5sJbTKEZ8tEGaOUxSlnciBJyI3BDkdRPSi2+0O8OS56Ic3Qtxz5GUiSgPwh2Bs
gKIoE4jIRwWTmX+F1gpdvx2UbLFYXsvNzV0pcBr+AvCCfi1Ai0opivIJEd2jHyOiWGYep99aYk3m/
TnB9tvbqqo+Q0SNPT8zMxtS0P8RJpOp5vnz57P9bcfNhuFQ6Jg4ceJWAFv9bYfB9UFRFKSmpi5m5vu
IqLBmPu1VVX3d++XshZlTJEl6D5pAlY9D4okA+FRZAIDgRgNoT/FTIb65zi9WrFj/
nJycTfAtEwQzbycivTNxmYjOQaCVAKA/
Eb0JceXDImYWac1fNJlMzRRF6Qigp+i6BHadBdAUQB8i0pfX/mUymT5wu92rBDf/
wZ4bq0jDA9BtL3jooKrqYQDb9dEEZk4iIh+niohiTSbTHy6X6w/
oqiw82xJvkaBUl5n3k6a4qX8fLhJRa5fLNVkfffIkbuq1Pbx0Z+YYUYIta+qovaHTRmHmLdAaqemv9
TvP9Y7Ld51UsmRJUSWQwQ1AVdW7ACgADKciH8aWhw6bzfa4zWZL8bcdBtcNWVGUFEmSZhUyPpCI7tW
H/lnrBvm2qqqz4Cu3DQAQhK0LhZkPyrI8AFr4Ws9SIorNyclZAnHyZhIAH7VGIkqFuPHU19CcDP2T/
wVoLbaFjWuI6AVFUaqI9vBFsCZ29awsy3WJaIhu7CyAbm63e6E+suO5kR+FJq5VJDzrLyGiZXoHwHM
jryKYNjAkJGR2bm7uEvgKma0F8CgR3S+Y5wAwDwKnSpKk1p4qGh8nrpDtMECrUklnXQmyh/
8joiegSzpl5q1E5BJc61az2TxCnxgsSRIURTEU8Zx6AAAgAElEQVSqDv4jVFU9VrJkSZEjf0djOBQ6
XC5XFgDRfrjBrUloqVKlyntKEgvAWivyk/pEQUDL6ldV9WtR1cY/
wGEymV53u93z4eucrAsNDX0FWrMrUeloChGVKcR5ESWb/gLgEoAXdetkA5imv/
Hnoz0RZaqqKpJ4FiLL8ksAoFea9FQktGetr4b+/Zspy/L3Ra268aw3JSQk5HNmXg5fdc/
CbuRfWa3WkQ6HYwb5loceB2AS5Gd4z/eV6H0iotdUVY1FEbeCPGutAfCNKH8EWu+ODAB6bZRjRPQ/
+JaNnpUk6WW32/0jdNEWVVWV9PT0s0W1y+DfwczFsrOzk/
xtx82GseWhw2w2HwAw2N92GFwfJEmKjIiI8GlvzcwrZVmeqarqasG0BFVVB4tC4f8EZu6gKEqCIES+
1Wq1NnM4HOOIqJlgqgOABYIul4Wwi5kXepIg9cwG0KeQeYMkSdqsquqWQrZqRLQDsBPADv0NXZKkjs
z8sSBqsISIBiuKslmUACmCmVeGhYXFOhyOpaTTobgCs61W61sOh+NTiLdULkLc/
RXQKkB66w+SJkL1FPkKdV2JowDegLbVpb/
eDdCcv5n5D3oiYx+KHC5ZllsqijJUkKeyXVGUaBhJmf8VpChKuZkzZxpJmTqMCIWOxMTErMTERKOX+
W1ASkrKw8wcLMuy3nHebTKZhqiqKtIc2Qig7XV0JuKg3cj17Wx3SZL0rNPp7EVEXQqZrhTVDk8kYyi
07Q49iws7BxFNkiTpW08L83DRawTn6mO1Wn9QVXUhdD06iOgtZn4Dvkmim4ioo6qq868h6vOX1Wpt4
3A4voe4mZrItpVWq7W9w+HoBnGbckC8TeTtIfKKPjESwAxVVUNI3JulMDsyTSZTcyL6Er5daA8BGCG
KWkiS1IuZfap/
PEnDDxPRG7qhoxaLJcZT5WGUjf4HDBs2jJi5qCJvdxSGQyHAZrPtjIuL89EHMLh1SE9PD8/
JyVkKaPvL+TgJoIOiKHOge/
Jn5l+Z+W6IEwKvGU9ZYXN9sieA3bIsN2bmZtDad4twQNBTQ4Qnh6CHZ4tBn8C3FYWH6JeHhIS87eks
KsolEJ1rXFhY2OcOh2M6fCtOxqmq+gh8Ky12AIhh5i+p6GWoJwMCAp5zOBzDISiZLMS2rVar9SWHw9
ESAiXTq/A7EVWHbw7L/
4joMGlCY0XGU9L5AnRbTwDOy7LcDcB0QdRilKqqb8K3kdpgTzWPXkX0gizLzwUFBZ0HwEWN+hj8O1J
SUuLMZvMOf9txM2JseYgp4cmoL/
I+r8HNhdvtnm4ymSKLFy+e36G4KElSK0VRphJRRd2UP61W60tpaWm1mfk3/XpENJSZm0JcUeEDM/
8E4BAR6XuB7LNYLM8oivIwdOHufK/
```

```
JIF8VzELx5CsMh29fjx+IaDwzLxMkRm6xWq2tnU7nGPj21CiM2WFhYe84nc6hzNxJN7bQbDZ/
43a7N+uOJ5vN5qZut/
t1wRww82qBw3WRmWNycnLeIqJ3dGMXmNkiKK88YDabY4goy+FwiPQpnCgkuZaZDwDIEVTRHMzJyflJ
luVxqqqCmaGqKgAgOzt7rCRJTWRZruY97h1zOp0TcnNzH7darYPyjzFzTkpKysTw8PDvTSZTmHdNZk
Z2dvYhZn7BbDY/kH9OVlZW0oULF6pFRER8AMC7DhRFUex2e5Isy/N/+umn3ampqUxG98v/
BEmSHgCQ6m87bkYMh0IAM/dgZqe/7TD45xDR4MDAwIeZOTI1NRWXL1/
G6dOnF4SGhk4vVqxY1fxljMx8KTk5eVl2dvbboaGhFcqVK5d/HWRmZm45d+7cXffcc0/9/
PNUVXXa7fZfwsPD2xBRcfq76+PxpKSk9S6Xy1ylSpX1JUqUeBwAFEU5l5SUNM1kMnW57777huVrqw4
iAjPD7XanmM3mArkWRIS0tLQf77rrruoBAQEFqj3S0tJ+SEpKKlG5cuWdYWFhD3tvULm5uft27ty5R
JKkStWrVz9usVge9Fwr3G73uQMHDszIycl5tWTJkqcqVap0UZblkswMALh06dKvxYoVq2wymSp6z3P
58uW9e/
fuXZ+QkNBz3759HU2mv786XC7XmYMHD250u93NQkJCJt19993diaiEoijpx48fn5uent7lvvvua1+8
eHF4z8HMyMjI2HzixImNd999d3jx4sVreMfOnj27OjMz851y5cq97H1PPTdRV2pq6nZZlouFhoY+5b
1WRVGyHQ7HBUVR5ickJMjz58+/EBQUFE5Exb0358zMzEuyLCtmszncO88zxoqiVJQk6QG3263/
GN0PwCeZ18OVIhY9CjkeAGDQqVOnRGM+jeFkWYaiKKUtFsvLqampMhFBkiQQEbKzs0+4XC5L8eLFg0
uXLt3R6XTmut3uoua/
GPwLsrOzhwQFBX1w9VfeeQjVc+50hg0bJu3duzdg7ty5Ro3xLUxmZmbkihUr9mdmZlJWVlZRe1EYGN
xSmEymZffdd9+jhw8fLhkbG2vs7d9gunTpUlZV1axp06al+duWmw0jQiHg2LFj94aGhu6H8f7c0pQo
UeIMM0dNmjQpEEC0JEkUGBhY8sEHH/zQbDY/
pCjKqVOnTo1KTk7eR0SqqqpsMpkUVVW5fPnyFSpVqtTv0KFDA86ePXuBiFS3261GR0dXL1eu3Fd2u7
3/
rl27fici1eVyqZUrVy5Vvnz5pmvWrJlCRKqnIZnidrtZkiS1YsWKJQ8fPpxmMpnIZDIRADz11FOdix
Ur9rHL5ZrqcrkuJCUlTT916lS6LMtUv379psHBwZ+53e79O3fujPWq8pUtWzYwOjp6jCRJob/+
+uurbrebZFkmAJAkiSpVqlQhNzc3NyUl5ZyiKHnHJUmiOnXqtHY4HHsPHz6cpB+LiIgIiYyMbJCUlL
QmIyMjR1EUkiSJ7r333gccDsdJh8ORLUkSAYB3rFatWm2ys7PP7N27d5N3rczMzMCUlJSv77vvvliT
yYT851FVlapUqfJweHh4p927dw/
NyMjIzj9WokSJwKysrEzyhCVUVaXAwMCgqlWr9rtw4cLCY8eO7co/
ZrVay5jN5oBz586l5D9OROS11ftz/v+PiIioXK5cuZ4XL15ccfjw4RXesYceeiiuePHiDc6dO/
fViRMn/
gQ0wahixYoF33fffV0DAgKqJCcnf2y32095x5iZQkNDI6OiotodOXJk0uXLlzOYmcLDwytGRUV1J6L
iWVlZ2w4dOjQLAJg5z56wsLB7zWbzXWfPnt0LgAIDA0uVL1/+DZPJFJ6bm3v24MGDX3k+yhQYGFi6X
LlynS5fvrwvOTl5pXcNAORyufarqvplhQoVnr7ef0MGvphMpn3QVGiNnk8GV6dHjx6BXbt2dXXt2lU
vw2xwm2C324fb7fZ/FLVwOBxFLeO8Kk6n88krjIX8kzF/
Y7PZ1LZt297U4feLFy8Kc1Su9PtwOByFlZoKcTqdNe12e2+
+BknsjIyMCKfT+YLT6fRRvUxLSxPmgBj8t9hsNldcXJy+OZ+BQeHEx8e/
2qFDByNMbmBwjcTHx0+y2WxGXwkDgzsMI6RfCIqiLAkKCioOIN3fthgY3EqkpaV1nTt3riEDbXDb0a
VLl7KyLEd5ej4Z6DB0KApBluXmqqru87cdBga3GqGhoZfbtm173baFDAxuFmRZ/hHA//
nbjpsVw6EoBEmSdgMwx8XFlbvqiw0MDAAANpvNDEAODw83vlsMbkeCVFX90d9G3KwYJUaFsH379vM1
a9bcfvHixX379u0zwrcGBkUgKChICgsLezozM3PB7t27M/
xtj4HB9WTHjh3f7ty501DJLATjKeIKmEymP61Wq1D338DAwJfffvvNDaB+8eLFi9rIy8DgliA2NrZC
XFycvjOsQT4Mh+IKMPMDzLy5bdu2Fn/bYmBwq8DMXzLzEX/
bYWBwPZEkaYokSfH+tuNmxnAorkBCQsLv0ERlavrbFgODW4WJEye+O2nSpN3+tsPA4DojKYryqb+Nu
JkxykavgsvlijCbzZn+tsPA4FagYcOGpipVqsxn5qETJ07c5W97DAyuFxMnTmzkbxtudowIxVWYOnX
qeQDde/ToEXjVFxsY3OFUqVLFAiAGwFP+tsXA4HoRFxfXJj4+/
m1/23GzYzgURWOoy+V6399GGBjc7DidTjcRrSCi4/62xcDgeiFJ0iRmNnLproKx5VEEmHkYEd3jbzs
MDG525s6dmwugpb/
tMDC4XnTv3j3U7XavmjRp0mf+tuVmx2hfbmBgcN1o27atJSQkZBqAT40cCgODOwtjy6OIxMXFtbHZb
Gv8bYeBwc1MsWLFzADaAHjG37YYGFwPbDbbydjY2Ar+tuNWwHAoiggz7yKiWjabrbm/
bTEwuFk5efJkDhFNkiRpp79tMTD4t8THx38OQJ48ebKRE2RwfbHZbM1tNtsD/
rbDwMDAwODGExcXVz0uLq6Mv+24VTB6eVwD27dvP/
TII4+UqV69urxr1y6jT4GBgY62bdvKDRo0GFe7du2k7du3O/
1tj4HBP6VLly5BzKxMnjz5rL9tuVUwtjyuEWZuYrFYkv1th4HBzUhERISJiHow83P+tsXA4N9gMplO
mkyml/
xtx62E4VBcI5MmTfqCmS916dLFUE0zMNBx9uxZNxGtAmA43Qa3LPHx8a8C2JeYmDje37YYGBgYGBgY
GBgYXJ24uLjONptth7/tMDC42YiPj5/UpUsXQwjO4JYkPj6+j81mG+vvO25FjC2Pf8jhw4e/
J6KH4uPjjXa2Bgb5IKIusiw38LcdBgbXis1mu5eIPk1MTOzvb1tuRQyH4h/y22+/
uZk5EsBGf9tiYHAzwcxLABzxtx0GBteK2+02Xb58uZS/7bhVMaS3/
yWxsbFWSZLGElFcYmKiy9/2GNxcMHMAgPb+tsPghrObiLb72wiDf07Xrl2jXS6XdcqUKev9bcutiuF
QXAdsNttWAObExMSH/
W2Lwc0FM5cFcMrfdhjccKYT0ev+NsLgn9GlS5dGJpNpNpRWJiotFR9F9gbHlcBxITE+sw87fDhg0z3k8
DAwODWwyTyRTjcrmi/
G3HrY5xA7xOpKWlTUxJSXklLi6ujb9tMTAwMDAoGt26dauYm5s7YurUqef9bcutjuFQXCfmzp2rANh
```

```
LRONsNpvN3/YYGORn0qRJOHDggL/NMDC4qYiPj/
9DVdWvp02bluZvW24HDIfiOjJp0qTdbrf7YVVVD3fu3Lm8v+0xuHlIS0vDgAEDULNmTVitVjz66KOY
OnXqf3b+IUOG/
Kvz+dv+q3HgwAEQEYgIwcHBqFatGjp16oRDhw5d81qqquLgwYPIzc29AZYa3CzEx8e/RETTExMTW/
jbFgODKxIfH98nPj6+j7/
tMPAvzFw2KyuLq1Wrxvfccw9/8sknvGDBAu7bty+HhITw1q1b+b8gIiKC+/Xr94/m3gz2X439+/
czAB40aBCvWrWKJ0+ezK+88gpbLBZevHjxNa01efJkBsBnzpy5lmnf+fuzZlB0OnfuXD42NraCv+0w
MCgSNpvtbpvNdtZms731xhtvRPjbHgP/wMxlf/rpJwbAa9asKXAHcjqd13Sj+zf8G4fiZrD/
angdimnTphU43qxZMy5XrhxfunSpyGsZDsXtTXx8/
Ns2my3d33bcjhhbHjeIxMTEE8zclJmbWyyWvR06dAj2t00G/
kFVVQCA2+0ucDwkJKTAz6mpqYiNjUVUVBTuvvtu9OzZs8Cc9evXg4iwfv16PP300yhZsiQef/
xxrFu3rsA6drsdnTt3RmRkJKKjo/
Htt9/62LR06VI88cQTCA0NxT333IM2bdpgz549/8p+i8WCMWPG4Pnnn0d4eDgqVaqEDz/
8sMBrFixYgFatWuH++++H1WrF888/j507d/
qs891336Fjx44oU6YMoqOjkZiYKLTtavTq1QspKSlYsmQJAGDPnj3o1KkT6tati+DgYDRo0ADz5s3L
e310dDRiY2MBAJGRkSAiXL58GQAwcuRING3aFGXKlMHdd98Nm82G8+eNPL5biS5dugQRUQ8iesLfth
gY/
BPIZrPF2Wy2B4y8ijsPZi577tw5joyM5MjISJ43bx5nZGQIH3GfeOIJDgkJ4REjRvD48eO5zP+zd97
hUVTdH/
+eO7ObhARCFgQpShNpdhCwgyhV4AWJCAhiykYULOjPgqhRsKAUwQKkAErxlYgoRREVEBCwgAoKilKU
GiCdlC1zz++PncRkdqJEX1k09/M8PMq9c++cXfJkzpx7zvfUr8/3339/2fyGDRsYANevX
T09P5l19+4b59+3K9evXy/
pA9rKH9nPHHAERo8eXWHsoosu4r59+1a678qVKxkA//
jjjxX2GTVqVNnfs7OzGQCnpaVVuk9lDgUzs67rfOedd9quO3nyJAPg1NTUsrFTPfJ4+OGHuUmTJqV/
VQ7FGUxiYuKbbre7SGkF/b2oL/c04vf7hwIoSUhIGB5qWxSnl/
POOw+fffYZli5dit69e2PVqlXo3Lkztm0LNKxdt24dAKB79+5lazp27IiTJ0+WXVPKOef8FuiKiIgA
ABQWFgIA1q9fDwC49tprK1wGRlZ9ve2bduiXbt2SE5OxiuvvILDhw//
ZftLadiw4svfhRdeWOFIw+Px4Mknn8QFF19+4vd4Wu6xPdbvemlJSUK0Ntj+L00qZNG3To0AH79+8HA
DRqFFD9/eCDDxAeHh507alSr149AIE37rPPrry4qFmzZpgyZQoA4LPPPsOgQYMwZMgQ/
srIzs5GkyaB6ryff/
4ZGRkZeP755zFgwIBTus9f5aOPPgKAModiypQp6NOnDxITEwEEJ5pWRkZGBnbv3o158+bhiiuu
VfcPcLvdFzLzFADvq6aNpw/lUIQIU+a1kdvtbh0XF1dT1/Uni4uLn5w/
f37V47qKM5qDBw+isLAQrVq1KhszDANHjhzBVddBQBl/w0LC0OXLl3+9L06dOgAANi+fzruP/
8snv93pv4VVddhWuuuSao2qIq9pdS/
j6GYWD9+vW45ppAQn1mZiaAgDNTyjfffFOVj1clfD4fXnnlFTRp0gTdu3fHvffei5tvvhnPPPMMAOCZ
Z57BoEGDMGTIEDz99NMAgM8++wyDBg3CkCFD8PTTTwMAPvroIwwaNAhDhgzB008/jaKiIgwYMAAA
sGfPHtx7770YPHgwBg8ejMWLFwMAZs+ejcGDB2Pw4MGYM2cOAGDfvn0YOHAgHnvsMQwaNAiHDx+u
8Ln+8Y9/4Pbbb8fAgQPx+OOPo6SkBAAwZcoUDBgwAIMHD8bMmTMBAMuXL8fAgQMxcOBADBo0CPn5
+RXuu6ysLDz44IMYPHgwBg8ejMWLFwMAZs+ejcGDB2Pw4MGYM2cOAGDfvn0YOHAgHnvsMQwaNAiH
Dx+u8Ln+8Y9/4Pbbb8fAgQPx+OOPo6SkBAAwZcoUDBgwAIMHD8bMmTMBAMuXL8fAgQMxcOBADBo0
CPn5FXLvfvWzzz7DoEGDMGTIEDz99NMAgM8++wyDBg3CkCFD8PTTTwMAPvroIwwaNAhDhgzB008/
jaKiIgwYMAAA
```

```
ft25dr167Nbdq0CSpFXbt2LV999dVcq1Yt7ty5M2/dupUnT57MQ4YMqbDPs88+W/
b34uJiBsDTp0+vtOKitMoDADscDm7ZsiUPGjSId+3aVeG6ffv2lX3utm3b8rx583jVqlXcsWNHLi4u
Lrvu5Zdf5saNGzMAfu6555iZedasWdyuXTuOiYnh/
v3788GDBzk+Pp5ffPFFZlXlETLuuuuuqNjY2Ai32x3rdrvvC7U91RlVvngGkpiYOJKIZkopR6elpaW
H2h7Fn4eZGwI4FGo7TgdOpxNPPfUUHn300VCbEgreIKLbQ21EdcPtdicw831E9FZKSsqEUNtT3VFVH
mcgqamp84QQ5xmG8Z0p4f1MXFxczVDbpVAoFGcCcXFxNbt06aIzc3siWqeciTMD5VCcocyaNevQ3Ll
zP8/
JyTkOoJ+macdGjhxZ+w8XKhQKxb8Yt9s9SdO0tBYtWjTKzc0dm5KSMjrUNikCqCOPfwgJCQnthRA/
AxjNzK7U1NQHQm2T4o+pTkcen376KZo2bVpWJlrNUEcep4GkpKSBzJzEzBtSU1MnhtoeRUWUQ/
EPIz4+vqOmaR8w89JGjRq5lfLbmU11ciiqOcqh+JtITEw8H8BcItoIYLzf7w+fM2dOQajtUgSjHIp/
KEOHDo2Jjo6GYRhfAXgvJSVlbKhtUgSjHIpqg3Io/
sfExsZGtGvXznPkyJF0Zr6OmQempqb+fcIlir+Mcij+4SQkJNwmhLjX7/
ffxsxFDofDn5KSciTUdikCKIei2qAciv8hSUlJz0opL2PmZ9LS0jaE2h7FqaEcin8Rbrf7dgCzmflV
lWNxZsDMDgA9Q23H6WTWrFkLmzdvPqt79+7V6UHwExH9EGoj/
g0kJCT0FkLcD2C9qt5QKEJIYmLidW63+/
8AwO12JyckJPQOtU2K6oXb7T6WmJgYF2o7FP8c3G737W63+7jb7Z4PgFSb8X8mSnr7X0ZqauqnAD41
/
1pLCPFuYmLifY0aNZqlEjgVpwMhRMfCwsLjobZDceYTFxfXXNf1owAuBJBRXFz8fwA4OTmZQ2ya4k+
gjjz+5YwcObJ2ZGSkVlRUVMPhcKwDsMHr9d6p+oUo/
i7cbveFAPaodtGKykhKSurAzGuISPf5fPXnzJlzEgHpdMU/
GOVQVCMSExOHA0gUQoyRUhYDaJGamvpBqO1S/
Ltwu90niWjk7Nmz3w61LYozi8TExDsB7BJC6FLKVrm5uXMzMjKKQ22X4n+DciiqKYmJiTcQ0fsA8lN
SUuomJycLdSSi+F/gdrtPSilvSUtLez/UtijODNxud10AxwDkSSlvSEtL2xpqmxT/
e1QORTUlNTX147vuusvl8/k6A8ChQ4cmut3uHlLKB9LS0taF2DzFPxjDMC7Jz8/
fF2o7FKHF7Xa3BjAdwCbDMF5m5jZz5sz5MdR2Kf4+lENRjXnttddOAvgYAIQQC5j5Kk3T7gGwzu12d
wVwJCUlRZXCKapESUnJsZiYGAHACLUtitPPsGHDGi9cuPAggEUAvADeSk9PzwaQHVrLFH83yqFQAAB
mz569E8B1sbGxGgAQ0Rhm7pmQkDA1LS1tfIjNU/yDiIyM/
NUwjFsBrAq1LYrTR2JiYg8iykBAyK0NgE4pKSm+EJulOI0oh0JRgYyMDAMAZs+ePTA+Pr5j6bjb7d5
GRDnMfEdKSsqvobNQ8Q8gmogKQ22E4u8nKSmpg5TyTiJ6GsDZzPx4amrqdABQzkT1QyVlKk6JxMTEA
QAeEEJMIqJthmHcCWCJ0tZXWBkxYkSdN954IyvUdij+PtxudwsADQCsALAcQJxyIBTKoVBUGTNjexO
AaCllZyFENoBoFblQAIDb7W4AIE/pUPz7cLvd0wHcJaUc4vV61yrHUVEeJW+qqDIpKSknUlJSzvf7/
Zfm5eUdAnAbgN1ut/ttAIiNjXWG1kJFqOjSpYsOYBczXxlqWxT/
G9xu92K32/0NAEgpT0gpz09LS3tbORMKKyqHQvGnmTNnzmHzf19NSEjYSkRNASAmJuYTt9vtAjA8JS
VlW6jsU5x+1q1b5z///
POjmdkTalsUf45hw4bVioiIGO33+6fquv4ygFaGYdwFAGlpaapZl6JS1JGH4n+OKav7GDO/
4fP5PnA6nbOY+bPU1NTUUNum+PuJj4/vmJ6e/iWUlPI/
hhEjRtRxOp21mLm2pmmbAWzMycnpUZqkrVCcCsqhUPytxMbGarVr155PRKgsGh0AACAASURBVD2Z+U
G/379c07Tr8vLylqpfVv9OEhIS2tesWXPntGnTlKTyGYzb7a6xe/
dub8uWLZcS0Y0ArpRS+g3DyJk7d+6BUNun+OehHArFaeWOO+44x+FwfAcgqqSkpJ7P5yO/35+jnIt/
D0lJST6/398xPT3961Dbo ggmMTFxABFNJaLHmXkzMz9MRFOViJ3ir6IcCkVIcLvdLbxe7yGHwzGaiF
5g5o2pqanXhtouxZ8jPj5+BxH5hBBg5kuYeQ+AYgCclpZ2cajtq+7Ex8efp+v6+bNnz37f7XYfYOYP
U1NTE0Jtl+LfhXIoFCEnMTGxHRH9JyUl5Rm3250MYKSUcnxaWtqCUNumODUSEhJyhRDR1nHDMLwej+
fRBQsWTA2FXdWZhISE+mlpaZlut/szAB39fn+rOXPm7A21XYp/L6rKQxFyUlNTvwfwPQD4/
f4puq7XFUJcD2BBYmLiUgARhmHcpX4ZnrkQ0c9SyvZCVKxEF0L8tGDBgmkhMutfRVxc3COapo1OTU1
t/HvXmUcacwGkAXiQmRf7fL6b582bd/T0WKqoriiHQnFGMWfOnAIAo0v/
zsypRPSkrutrADR1u93JUsoDaWlp6aGzUmHFMIwZmqbNBFCjdIyZi4loBlS1x19m5MiRI4UQ9xDRWd
a5Ll266K1atZoOoMXs2bN7EtFVzPxaSUnJMwBQKoWtUPzdqCMPxT+KhISEeCHEDAAZKSkpI5OSkgbO
nj37nT9aFx8f//3OnTs7bN68WVUe/
E0kJCTsAHCBmUcBZv4+LS3tglDb9U8nLi6uuxBilhCiGTMjNTWVkpOTxeHDhxNSUlJSEhMTbyGiGcw
8KjU1dWmo7VVUX1SEQvGPwoxMpI8cOfJsAJBSDnW73W9LKZ8SQjxbVFRUd8GCBUfKr4mPj99DRM3bt
Wv3dZMmTXr+97//3R8K2//
tSClnaJr2EoAaRFQkpXwx1Db90xk+fPgFQoiXhBDNAICIC4Ha7jxw+qDgKgXbgKYZhfDBnzpyzQ2upQ
qEiFIp/AV26dNHPP//89gB2AMgDkAvGq/Xu3/evHkliYmJx4moLgAYhrHL6/WOmD9//
lehtPlfCrnd7u1SygsAfJ2WlnZZqA36JzN8+PB6ERERqwFUqJKRUoKIhqSmpv43RKYpFLaoXh6Kfzz
r1q3zp6SkfJ6SkfJ6SklKUkpLikFLeA8DndDq7ud3uXAB1S6/VNK1NWFjY4pEjR/YNncW/
kZ2dHZ2Xl9fxj6/8jfz8/Dp/lz1/EZZSpgMoIKJnQm3M/5qcnJzaNmNxubm5/3PHKTY2VgsPD/
8IFmcCAMzEVyWBrTjjUBEKxb+axMTEHlLKVZqmVRiXUh70+/
1PXnHFFQYz1yGikwBOMnOhpmkF5557br1GjRp13LJly2uGYRQCKEpKSsoDgNzc3JjatWvnO3Mz
kpl/drlcG8uPZ2VlzSWin4hIi4mJqfAgyM7OXgbRiXUh70+/
EJd148YhtEuJibm09JxZq6Rk5OzGcAPMTExdxBRWUfPnJyc2jExMbl230NOTk5/
KeVtLpdrJBEVWua6MvN9MTExtxJRhRyTrKysc3RdPzs6OvpL657MHGG93lzTmIhuc7lcz1vn8vLyLv
f7/T/WqVMn3zLuio6OzrazvbLv+1TIzs6+xuVybbCOV/
```

```
ZdZWVlPUVEJS6X6znLeCyAeSUlJfeuWLEiA4Cjc+fOV9apU+c9ZobP5/
vu2LFjk3bs2LFdSqlLKfU2bdp02L179/dSSl0IoTOzzsy6EEK/6KKL7i8oKNi5d+/
erxFIZC37w8yRJ0+e7G0YRp2CggKZk5MDr9frRLkXQGb2pKWlhf+Z70Sh+LtQDoXiX01iYuLPRNTCO
i6lhJQyq127dnsiIyPDiCiKmSMBRBFR1B/
tS0QeZi5hZj8AnxCCatSoUZ+I4PP5sj0ezyEAvho1atSPiIhoJIQAEcHv95/
Izc3dahhGSXR0dItatWpdUDpnGMbxo0ePfiCEoEaNGt2saZouhHAWFBSsP3z48HoiMpo3bz44IiKiN
QD4fL6jBw4ceLO4uDgfAFq2bHkHAD07O/uTrKysshLbyMjIs84999xEIYTD7/
cfO3DgwJtFRUW5ABAdHd24cePG8QDg9XoP7t+/f6HP5ysBAF3Xw1q0aBGvaVrtzMzM/
5bfEwCaN28+WNO0sP379//X5/
N5zTWO5s2bx2madrZhGJ7MzMylubm5B4lIi4iIqNugQYPBhmEUHj16dJXH4ykEIFwuV7uoqKj2J0+e
3Jmdnb0LgGBmjYgEETnPOuusa7xeb0FBQcFuv99vEJFA4OGq1axZs63H4znu9XrzAWjmuNB1vZau69
EOhyOmuLg40+fzFQIQRCSY2RkeHl5PSlns8/mKS9cQUZimaRHMDMM4c4Rb8/
PzF+3evfsjZj4LQGMAZwGokZaW9p8Qm6ZQVEA5FIp/
NfHx8SeJKJKIwMySiLKZuQBAJhHtuPzyy6cz82fMfJSIjgJATEzMxbqul4W3mdmflZX1udPprFmrVq
2LSscLCwv3FBUVHSQiERMT00HTtIjSOb/ff7K4uPhIjRo1WpoVD2V/
DMPweTyeE2FhYQ0s416fz1eo63o0Ag/
VM+rB9i+miIik0+mMEkJACAFN0+Dz+bKKi4sPxsTEtNV13aFpGogIpgPoNx2+suuJCLqu48SJE0vq1
q3bS9f1Gua1eQcOHJh64sSJ3eedd96V9evXH0NE0DQNQgh4PJ55GzZsmNK1a9fEyMjIe0r3FEJ8u2z
ZshwpJQEYk5SUtCPUX5RC8XuoKg/
FvxpmLhRCfEtE33m93i2apu04ePDgrtWrVxcCQIcOHd4EEC2EGOt2u+dkZ2fPBHCdZZs8IcTThmEsI
argg7cQQtwopXwUwDWW+0oialmJWTsANLEZ/
wKBt+Urf+8z+f1+MAekHaSU8Pl8b2iaNsLquOTn548LCwvrp+t6Z2aGlBIA4PF4Nn311VdPdejQ4WV
d188vv3dRUdGyzz//fNoVV1zxQHh4+E3l5wzD2L1+/fqk9u3bj6xVq9bt5ec8Hs9Gv9+/
r2bNmsOBQDWC5fs4rGlaw/JzJSUl6WFhYd00TWtq+Xzvrlu37v+uvfba/
4uIiHCXXyOl3HfixIk76tSp84DD4ehr2e9+XdePO53OMoXV0jkiOqbr+iPMPMfmK53BzPfY2PwzAI2
IHDZrKvvdOZ+Z2xJRjXJj0c2bNx8FIIWIxlgXMPOmW2+9tQUz32OZuljTNI+UMgxAkAqpQnGmoSIUi
mrN7Nmz5wG4nZlHxcbGNiKi8TaX3QXgUQDnlB9k5ulCiE+Z+Q91MMqt+YkCT67zLOMnARwmovPtV1b
KNgBVTQp8F4AXwC0WG1bXqVOnR3Z29sMArPkPexwOxxU+n28ggFnWDU0Rqwjr+O/
wFjOXENHt1gkiullKGUNEaVXYDwDWMnNnqx3MnC+EGCGlfIOIalnWvAGgFwLHCH8JZn6HiDwAhlRh2
UxmfhXA50QUaZnLefvtt/
OYuamU8upRo0Z99ldtVCj+TlSVh6K6UwIALVq0GGDnTBBRMjOPhsWZALBf07T5Usp5VbjXL0R0DBZn
wuTXP+FMfIPKnYnjdoPM/BUzf4tgZ2IXgNicnJw4BDsTx4UQ3f1+/
zWwcSYAoIrOxGcA1tk5EwAeIqL8P+FMfAbgQjs7NE0byMzJVmeCmd9h5k6wcSaY+aDNPWwTXk0+JaK
vUDVnYr2u608Q0TIbZwLM/GVYWJgTAK6+
+urHq7CvQhESlEOhqO54AKBmzZrdbebekFL2IqK21gkiekBK+bbNG29lZANYD+Aqm7nv7O7xB2yA/
bFJqXNg98a9H0AqET1pGT8hhOhNRNcxcwVJc2bOB3CDlLIRMy+poo12tv1ERI8DmGkzPcvhcKwwDKN
K92HmL5i5HsqVB5dCRAOklAkALrFMrSKiekTUymbLHURk7ZeRx8yV/
b7cLoR4BcCzVbD5oNPpvMXv9y8G0LwSG7pHRUU1BIDw8PAeWVlZXxQWFjY41XsoFKcb5VAoqjW1a9c
+G0BZTkI53gJQl4g6WSeYeY553t30VO7BzMUIvNkPt5nbDaBK8tTM/AmARgBibKa/
BuCyGc8D8BiAl2z268vMjZh5sXVOCNHPvGZFVWyshOOapg1h5nk2c+87nc6nfD7fyio4aQDwHQC2y1
dh5gQpZXMAt1rGtzBzEYCrbdZsYeamlrGTAHIrsWsfgLsNwzjl3jLMXCyEuMnr9T4GoKvNJesAXAgA
uv5bqgYRtfD7/
UGRDIXiTEE5FIpqTU5Ozg8AyhIWTVYAkAB62yzZYZaVWhM3K0UI8Swz32sdZ+ZDv3PMkVfJ+EYiCof
9W+3XAHQiqm8zdzczT7HJL7jV1KVYae5bBhH1B/
ALM6+u4kM+CGYu0TStj2EYkwCca5n+Vtf1ER6P5z0Azaqw7V4EjpGCnD4AjxHRMSKaYrHjRyL6mogG
2qzZZEYtalrGf4B9NChPCDEIwNyqfD+apo2QUl4OwC5B8yMAXUr/XupQmIm2N0VHR/
98qvdRKE43yqFQVGvMJDqUlJTsBQBmngtgF+zPwnOY+WNY8g9+D2b+Pynl3TZn5NkAbBUvmfkobLL6
mflHADmwOTYxIx2/wnyztcwlIJCbUKHfAxE9AeAz8zNFW+ZGMvOnzPxxJQ5KVZAA/
iOlvJWIullsy3Q4HN19Pl8qEVVFMTSLmdcB6GMzN0vTtI+tERdmPklE7wEYZbPmW2bWEeyofUpEHaw
XmxokvaSUk2GfE1MZzxLRL0SUajO3joguLT/gcAQKTI4dO/
ZYnTp1NlfhPgrFaUc5FIpqDTOXAMC+ffveZuZeRPQlgP+r5PI3iej+Kmw/iYiGWR/
kzHySmcOtEQFz7jCAIIlnZj5GRGsBBEmGM/
MRIcRyAP1tbHiOiP4D4CLL+CJN014DsIaI6lntrl279lvMvBpAkCjYn+AuIUR9Zh5rsbuIiLr7fL6x
RDTgVDcz180jojib6TeZ+WXDMFbZRFzeAPCQzX4/
AfjWxqHZxsyX29lARIOIaATsjywq430iSjUM432bufUI5L1UyAMpjVDs3LlTVXgozniUQ6Go7pQAgG
EYEEKEA3jN7iKzNPSuU93U1Dtoi+BkQACARaegdE0mEUVYH4TMnG+
+0d5ps1WOEOJ5Zn7AZu4tAA4AN1nGN+u6fo/f719tk3uwyOVyPZKTk/
N2FSMGlfG8pmnbmPl164SmaQOYuSOAh6u454t2n5eZPySiRwGshiW/
hJkzmPkOmzWZAN4EMMIytZeZG1by75SIwNGM3b+HLcz8U1hYWIKUciWCk0e/
ZmYfgHbWdR6P52tzvfpdrTjjUT+kiuqOBwCioqLOlVK+Wck17wHoUYU9lwHIh000AUAe2Ut755i5DE
GJlkT0PAIJlUEQ0X3M/LTN1CYAnwJ40DK+LywsbJDf78+ApeSUmde4XK5hWVlZc2B/lFBVFhHRLL/
fv9zG7ngEkintQv+VwsxPVSIOtTUsLGwkM68gokaW6c1E1NUmf+SkEOJZ8+in/
```

```
PhRZvZYI0vm3EQABwC8XAWb8x0OR1+Px5NqU82zl5m/th4FmSz/
5ZdfMgCAiDSbeYXijEI5FIpqTemRR/
369QdUcgTxDjNfb6cTUMl+W4hoLRHdZzN3GPa5EYVm2D0o0dIssRxrHTcZJqUcb7PnHiHENFiiLcyc
r+t6r5KSkpdgCdWbpaYDsrKyniaioDf5qsLMW51O51gp5Qc2ORjPMfNXhmGcsiCYyWQiGobgKpb9AA
Z4vd53EFwx8x0znwubklIhxBhmTrbYXUKBRm5tbO4/3+l0LgIQVA3zBwz2+/
1xCHbSTjDzQrujG2bequv6HY0aNSpNUlUOheKMRzkUimpN7dq1wwCAiMKsc8ycAeAGm6x/
W8ykyanMPM1m7gciami3jog2VHK88KKU8jbY6yvEMfNImyOLbAD3GIYRJDGtadrNfr//
XiKKtUwdCA8Pv4GIhpgOzF/CFIUa4PV637Q+mM2jh5nM/
FElkZrK9kwH0AHBCZDHNU3rDmA6gCsscweY2WkTsQARxUkpx8ESESKi1bDIqJusdTqdD/
l8vverWPHysBCiHiy5G2YeyMRKvu89Tqezl9/
vn9OwYcNEAGjXrt0pJwIrFKFCORSKaktBQUG91q1b3w8ElY0CwEIiutHm4VGZAuVRInoQgF2viO+Jq
HUl65YA6GkzPo+Z21civPSolPIiIrrRZm4EgBlWJ4iZ3cx8BSwVDsycz8y9vF7v5ahEBbMqMHOxpmn
9iehZBCcsbnQ6nXcT0fs2iaC/
x7sU6KfRxXKvQiFEL8Mw7JI6cwDkV1KWO0ZKOczGGXsPQD+b679zOBy3mmWtTatg938RqKIJyh8hog
eZ2U7m/YQQoofP55sAoF9pUmZ0dHRcdnb2X/
73USj+TpRDoai2SClrhYeHtwSChK1eRSA8ba22+JmZg6IVZjliPDOnWt+6mfknZrZVtASwjIhusBn/
gIg0IrreZm4mEWXaHakAGArgEVgqM0wtBrLLtTCFq6KllP+txMYqIYQYZhhGfwC3lR9n5p90XR/g8/
neQtWEvDZzQCrcmjRZKqndC/
bJkQdhk+QIYBIzX2pTvvoRM7e3XszMx5i5t9frfbWKSapfOxyOpxHom2JlNIC7YYk8MXOREKKnlHIw
gCSgog6FGT1RKM5YlEOhqLZER0f/
nJ2dXSZ8ZIbqH2LmOFicCWb+kZlr2OVZENEQZp5qk8S3h4hclYT21zNzGwTnP3zHzFtho6qJQLLnm2
zTMdMMnd+EYPXHFQDWE9Fs6xpmvkXTtCNEFCRq9Wdg5v8DUNOa5AjgBBHd4Pf7p6FqZZZ7ALxGwVLh
ADBcSlmPmSfY2PEl7PU4MojoqE3OwrcUELRqbLm+kIi6EdGdRDSoCnYf13X9Vp/
PtxzBTsNTzNwfNs6Opml9pZStATxTbgwAkJOTMzcmJqaqOScKxWlFORSKas327dvfB4CSkpJjQogXA
bxgUw2wGwHp5aAcCGa+FcA469GEWSkgYS9e9TUAYQ25mw7NbLLveLpZ1/
WHACy1mVvMzH4EIhQVPh6A2cz8ns2aR8PCwjb4fL5PYKN78SeYrWnat9bwPgdkpnuZTtptlawNwiyh
fYCZU2ymH0PgSGO+zdwmIgrSjmDmT4QQC2zyW/YjULVxsXWNpmkDKCBqNe5U7QYAIcStPp/
vdQRHitKEEDUrOaq6TUoZDmBB+cFSh2Lnzp0fVsUGhSIUKIdCUa3Rdd0AgOLi4gPMPN3mkgNEtIXse
3okIqCaeYVlvAjAQZszeiCgZvkjLJEEZi4QQoxn5hdt7rOLmW/x+/
1LEOygfI1A5OI5y5pMAI8xs10pbEpMTMyrXq93tfWt/M9g6j/
MtqvaIKLBzHxBJVGGyvYrIqIEKeVsq3MHYKamaZ+YCbNWNgO40mZ8OxElSykXWe5TwIG+KFadDpiVL
gYHlFOrwm1SynuJqLNlfKWU8gu2iHuZdjyoadpuu88kpfzEvEa3zikUZxrKoVBUawzD8ANAVFRU0Pk
5AhUTs2Bzfk9EjxPRxWTfE2Ij2cs1FyBQcnirdU4IcR8zT7YRtToqhOhFRLNgCZObb/HJzJxqGS/
SNG00gDk2xy3LXS5X0jfffPN+amrqhceOHbMxv0p853A47pNSrrTJH3lACFFSyUO50lbgQoghCEhUV
yg3ZeZ3dF2fbhjG+zZRpK3MbJcz8SsF2qQvsin9TTP1MKw8q+v654Zh2EWDKoWZn0JAFr2fZfxLAK8
SUVC0hZln6Lr+nvmZrCJaXxcWFr4JAEII5VAozniUQ6GotjBzRNeuXYNKPM25osoy8YkojZk9CCTXW
de9Q0R2rdAhhJiCYKEpENE9UsoHEXzenk9EN0op74WN0JSmaXdLKVNs3uLvk1K+AEsLc2b+KiYmZnB
2dvZ/Fy9efPXDDz+M+fPtTg1ODWY+6nA4bvH7/
YuJqIFlbg4RfWwYRlBSIjMfYWZvJdsOZ+Y7EZwD8VlERMQ9fr//
AwTrUOwA0MCmIidPCDFQSpkG4ByLDTNgJj5axjMcDsd0n8/3QVXKQ5n5bSIqQXCfkF91XR8DYJHNsk
VOp/
MZwzA+RHBp8F4ABWeffXYaAGia9pdzXBSKvxvl9SqqLTk5Oe2dTqddC+sSIcQwZn7R5mH9HjOvgf0D
4n1Urqj5DDNXcECOHDmCdu3aAcAMuwVt2rRZuXHjxivIvn/
IXVLKZOtbvHm0MBrBXTv3OJ3O3tnZ2c8Q0eChQwPpFqX/
rSrMXKTrej+v1zuNiC60zH0khHhaSrmFiGrk5eXhpZdewpo1a/DLL7/
IJk2a1Ha73RHDhg2zbvsQM3cgol6W/
X4EcGtxcfF7RNTMMrc7Ly8vsqCgoOE551TwGQCgj2EYjxKRNfq0gIj6AbBGBDZpmpZkHgVVVplj911
8CeBDANZI0UkiGur3+18nImueyioiutvn861BsKDZCQSqVK4tzaFo06bNWA6IYJ08VbsUitONcigU1
RaXy7Vxx44dTwGocL5PRIOYeRQsIkrM/
LlZLRHU3ImZPyGidjZhdSBQhtoXlrfQWrVqLZw1a1ZMw4YNex8/
fhzx8fG499570a1bN2zcuDG5ZcuWPxCRXTnnC8w8kIgqlF8y8zwANyK4EdgJIUQPr9d7W6lz0qhRI/
zf/1XWA+2PIaJb/X7/nURkdaC+AxDHzB8Q0dnFxcXo3bs3iouLERsbm92kSZMjP/
74Y7vx48ejbdu2uPTSsuaarwI4SUTWNu/
HNU3rI6WcDcDqGPxCRCe7du162Y033ogXXnihbEIIMUhK2Y+Ibi6/gAPtwc8joqaWvX7Wdb2/3+
+fY3dc9Tv8QkTTYONgmvkjT9gk7H4BYLBZdXKpZa6QiH4AcC3wW1Kmw+E4PycnZyoAdxVsUyhOK8qh
UFRrNm/evAgVHYp+RNSemSscMTDzbrPqYBURWbfZRkROAHZvtSsReAO1PuQ/jYyM3HXLLbdMBIBff/
0VANCiRQtcc80191933XUr/X7/
VutmZuJefat+henQ1EVwsmehpmm9DMO4jIim2n8LyEIlrdTtMDUwLgEQBwAnT57E66+/
jk6dOp3o2LFjP8Mw5sHUmvjkk0+wa9cuZGRkoKioCMOHD8fPP/+MgwcPIiMjA0VFRRg2bBiWLVuG
S5cuISMjA0uXLsWwYcNQVFSEjIwMrFixAsOHD8fPP/+MgwcPIiMjA0uXLsWwYcNQVFSEjIwMrFix
gOtsTBwjpayL4M6iOxDopWKtssjWNK2Xz+cbT4HOrKeEGYF4jJnTbH4mRgHoZT3+YuYfhBA9mPlVAE
FHY0T0BcqV1gohIISAz+fL1HX90VO1TaEIBSqHQlGtISIfAOTm5n7JzD0BsJlcVwYzHwNwh/
```

```
lGaU08PMjM38BGrtnUk8gFYA3h76JAK+2J1jV79uxZRUSvG4bxAZlql/
Xr18d7772HRx555GDTpk17zZkz5/
YVK1ZgxIgR6NixI5o0aSJ79uzZZvv27TdZ7oM+ffosbtas2bymTZsu7tu3L957L1BBumXLFrhcLqxf
vz4TpjMRGxuLxx57DC+88AI6dOiAli1bYty4cfB4POW3fW306NFt+vXr93S9evXgcrkwYMAArF692p
g0adJEKeVTVE40qlSBdN++fR+hXNloqTMxefLkfS6X6z+HDx9+u/xNhg8fjfjvPOO+8gM9/08ccfx/
Xu3RvNmjXDpZddeittuu823adOmrRkZGGde5XC788ssvSEtLg8vlQnx8/GYA+wDMeuutt9C/
f380adIEXbt29S1ZsmRneT2J0s973XXXLTnnnH02tGnT5t7HHnsMhw4dwqBBg9C0aVVP069cPP/zwg/
WfqZQHzETaCkcnzDzNTK4dbRk/
FBERcQMzj4N9Ce062O0h0aJom9+3bl1arVq2sygxRKM4ElEOhqNYQkQSAgoKCA5qmnWDmCo2fOKBeGI
tARYA18bBQCJFqI5QEZj5IRJsADLOMZwohHpVSvmJnz/
Tp0zOklEtg0TB48cUXPR999Y6+8EtWlSxece+656Nq1K5KTk/
MffPDBTTVq1GiYlJRUQUJJ86NChi7ds2TJ8yJAh58+cORMtW7bE1q0Vgx5O7NCDsb8+fPx7bffYsGC
BZg2bRrS09MxZ06Zjtb7vXV3dmZkZCSNHDkS33zzDUaNGgWfz4eHH354yOFi12wCHJ17tw5OzIyMn
/q1KlDli1bhsLCwvLfxe7atWsPB4APP/yw7KjI4/
Fg1apV3vbt23+fn5//L333Bmz+3Hsn5/fn6elru3XrNpGZF69duxajRo2CEAJTpkxzENGusWPFAorvCfYqKijYDuOKhhhx4aCZtox/
79+4t//PHHVpGRv6VoXHjhhSUAlhDRHa1bt8bgwYOxd+9enHfeeSgqKnr8o48+uu/
222/3Tpw4sQYA9O7926lBcXHxXth0N23atClef/116LqO01lKKK/3+/ppwvx/
PPPu1q1atUWwKUA8OSTT6Ju3bro0KEDXC7X8C8+/+Sd2acbt27dCqVSu0bt0akyZNwv3337/fln/
RkpZ57rrrrNWMgDAt7qu23ZQrV+/fln/
CAAIDw9Hfn6+RwgRJ6V8W0pJ5e1wOBwThRD+tm3WjFmLkQgWjF8y1atIhatWoV5s+fjxtvvBGffP
IJOnfurK1ZsyaFiDqURgO8Xi8vvT5OhP4fIY4ZVmOruNFmjVV3b17uQ0UPmZzMfnQVvP/+W3eJ/
Z2aBSw9eVTr8cKFj/+0pBxjLlikbWQfAgmqObZ5uZVhMGRQ3PWFbKoHV8qbS3KuhO7UQK/nmZ3jjm
XX8mz19oi62qzK2C+TKeK3UpQOvjWg7X4ug3RHbROa7yYOiSqIDPAxuVXBrMJQDMaWtZaWMIYhlH6
uT5GQFm0wr8TMx8UQjxjGMYbNl/rQCllDSKya3u+n5nrwl6tNJqZVzmdznColz/
FPwDlUCiqNbqu+wFA07QKmgOmnoQLwBDrGmaeI6W82Jq9b/
KqqaNgZQwCDkiFN1EzubBXXFycXctsABghthNAaNGjgajmGxt7cMuWLZeMGcz0P//
8M5YuDwgw5ebmro+Jibl/
9OjRUwCgdu2vRs9hGdu2g59QOZo5wOBxnWSfkWLKrVrMHHrm7r9rs9zwCTkiFwe+++LA/
CFDhpwtpdR79nZsCmyDh0h2fvoZCvi4KFp2kdJ8mdQCAp8c0330SzZs02fPXVW
VnNiSh8vp0ipL2dvL0oWl9q1a4dzzzzz3XHixO8HDjw/
KD16vVbHuu55547MSMSjI2LDhg3OZcuWjdcmWb4Gk33aatRg/fnzZ91e7du2Y2rVrl/
VZiY60Ro0aNXxdunTwD5ZdmyZSj/
Z9q0aQedTudArAr9e7KDw83K5pG+rUqWPAXXWCyLiFbARgyLiJKI6AUAMmv1WpfvnllyCA1J0wYCKDAPq0bt0aZ511Fr
Zs2YL+/fsDwC8AdgkwkwqvjLvq37RExMTCLVHmUp7HvFYWmKWAGH+/7f71+YmZmJTp0C7VGYOdPhcHxYUlj
dM2aNXr9+vX0YRuXPn0v2mIOCYdACATp06oXPzti+fTsADJJj63zigYrD3i+Ig6
pVtDmapnX1eDzPW8tuAaCoqOid3/4Y/
EwUE9EO2CdhBkFE8Hq9W8hHbn+YeajAMr/
wp8C4Cmby3OEEKN8K5zi5xXEkKOklHbn4N8zc5iNgsaX35p8gszF110U
UX1axZEwsWLECdOnWlYH/
evf0AoEaNGhZGzgyGhpcJWdrlcFR6G4+GhJzZDDP4I4I4hoCjPXdrlcFR6G+/
fvz87Nza29ZWjPz8/MDAI/MiswsDC0atWqNWqNEoyi5nDASSWXv/
```

```
aa69hwYIFGDBgwObCwsITQ4cOzUhJScHx48dxzz33lGpNDJ08efKcAQMG6HPnzkWfPn1KI0az33rrr
eYzZszoGxsbi4suuggnTpzARx99hIsvvni1EKIBMz/
Rrl07bN68GRs2bMDx48ezBg4cOHfq1Kmv9OrVC08//TSuv/56HDt2DA888MDRxx9//OG4uLiRNh/
5LafTGdSltBxjrANmefHNZFEKBfBBTEzM2Ozs7I/IpkEcEa2DpZT4D/
AfOXLk41q1ap2owhqF4rSjHApFtSU6OvrLL7/8sguAH8wH2GIACXZql8w8REo5GZawtlkeepyIuliW
7NU0bZI1499c83SdOnXSs7OzN8CixllUVHTEVEqskHzo9/
s9zLyHiK53uVxYuHAhnn32Wdx2221g5n0vv/zy8qysrHFfffUVkpICUfnRo0fD4/
F4Z82aVez1enu1b98ediWQNvYtRaCL6QLLVHbHjh3jLrvsssWPPPJI2XckhMATTzzx/
b333ruJmSskJU6YMAEej2fXggULosPCwtK2b99OnTp1wurVq9G2bVsIIW6XUr547bXXNqlfvz62b9+
OcePGAcBCIvohNjY2yel04t1338X06dPRrFkzDBo06MPnn39+GjOvAoCJEydi7NixGDRoEMLCwviaa
66ZfPnll+Ptt9/GhAkTkJaWBl3XjzRv3nx/
3759E2w+8rsAGui6HlT1cuDAgS0AOlNw07Z0AG2J6HrLkm+Z+dacnJxFFNxxFAh0hu1nHTS1SewqiZ
CXl/elx+OprPeJQnHGECTvplBUJ9LS0lyGYWRFRUWt7dWrVx0EK1qCmZOI6FoEa0oUEtFaBLe/
ziGiO6SU80vFqcqx0OVy3Zadnb0IwQmfJwCkI3A0YGU5AvLdVm5CIBdqmY3dxQCeIKKgluiVYUZVxh
PRKuteRBRrijaVvXUfP34cjz/
+eHZGRkatH3/8Ua9b19rjCpuFEH2klGsRnFMxDoGmZ1cBwKxZszBjxgxs27bt44iIiFeYOaixGAKOx
mQp5WdWQSkiGsnMUxCs+vkCM79r6oJYP+8nRHQYFv0Mc24JBWS4m1jGPxRCfMfM1mOsA06n83KPx/
MM2XcxXcnMXa12A/
iamc8i+1byoxYvXtwdgLjzzjtPWcVToQgFKilTUa0pKSkhAIiMjGwJe2eitHFXUCcrIvoYwc5E6YNt
utWZYOZPTGdiEizOBDOXMPNM2DsTy2DvTNzJzFlWMa5ydjxIRFWRa95PRGOJ6C3rhBDizvXr179088
03t/T5fGXjdevWPSqEeI2Z9dzcih3JmfknXdf7SSnnI9iZmM/
M18F0Jg4dOoRXXnkFgwcPPhoRETFBShnUG4OZP9E0LVlK+aHNQ/
kuKeVDCHYm3tU0Lc1MqLTu9wUCglJ2zsRH5hGWVU59O4CPrc4EM+czc0+Px3O3nTPBzGuY+TIbu78D
UGznTDDzdABvAjBsqkcUijMOdeShqNY4HA4CgPDwcLtf6B8RUTqAb22Wvg+gv3WQiOKllE9QcLfK71
wu14Ds7OxRCO4xASHEq8z8uJ0NALpbKz0APKfr+lq/3/8FALvW1ncDuB/
Brb4rI4+I7mDmBQCiy08w84MA3C1btjzvs88+w5QpU3D11VdDSlm4dOnSlJ07d45r27YtmjevcGJwX
NO0Hj6f73EisrZe3wAgmszGYgkJCfjggw9www03FD7yyCP9mfl9uwevpmmJhmGsJqJ65SeIKJmZ+1h
LchF487/HMIx1Nt/DDiHELGaeYxkHM39Jgd4sF1im9hDRSwgkaFZA07SbpZTXArD7N/
wSwNlWpVUABwBsBBBUZszMHzocjll+v39bnTp1fFlZWces1ygUZxoqQqGo1nTp0qUbEMgDKA8z/
+RyuW4DsMRm2afMbD07B4DxZtloha6YzJwZFhbWIzc3tyuA12zWpTCzXYj8ayK6zHp+j0AC4XSfz7c
Wloe/
eb8kZh4OS37G70FEI8yOnhWqFcwITXcAVzVo0ACLFi3CihUr0K9fP9x888361q1bH+rWrZv+7rvvln
2H5lFQDyllTyK6x3KrX82jmLI8gu7du2PRokUnFyxY0NXpdL4OS5SBmTM1TRtoSpJbP9MCADURODop
v+YoEQ0kovcQrAi6D8DDds4EgD0I9F+xNh3LEkI8IqUMKu81I1I1AMy0zpmlutJGf6QAgWoiO2dilx
DiYb/
fvx5A86ioqFZhYWFBvdkVijMNFaFQVFvy8vJanjx5Mg0I0qHI0TSth9ku2voA2wHgQrskPSKqD7Ncs
tz4SSK6saSk5FwAb9lEGpYB6IFgpcQ9CCSAWkP4n2qadpfX611HRA0RzKNmoqBdQqAtzJzIzI9a9Sm
YeQmAWlSuY2a3bt3QrVu30khMYiWJh4OYuQEszhMz5xPRfrJ04LzlllsAoJff73+GLHLWzHxSCNHTM
IzpMLUwys2tBrAeQIplvEgI0ds8MrCuOUhEIwC8Y/
M9HCOiL4jIehx1kohGGIaRbj16YOanAew17bDudwiBhF87zZInmflZm5+H4wDiTRn4s4DAz2Z4eHjT
7OzsXi6X64OgnRSKMwQVoVBUWzRNywew2/z/
snEiGsDMXRGcN3GAmesguMzzQwA7YVNaqGnazZqmlRDRSptIwyYEzuitSX+ZAEoANLWM/
+RwOGL9fv9iAEHiSwBeAFALwGDbD2wDMz9FRP+xOgZmcuZBIhpps2wcM/
e1cyaYOYGIDleS15EJ4FrroNl8bRgFtxWHGZm4D8FlljuEELOIKMW6RggxjJkTEVxNcVzTtP7MnIJg
+fNCIloKG2VU08F4zqonAWABEb3NzHb5GSeJaEUlzsRDRDTOxiktFkLcCiC9/
PGIEKJU+juoCkWhOJNQDoWi2hIVFZX57rvv9gF+O/Igs3OoWRZYnuMA/DZRgW+J6A1TgKkCRBSn6/
p2wzBWI9gJ+RGAhCVZkQM9MA4CaGe9v5mT8JKNPgUApDLzbgTUGU8JDjS5agrLcQEC0ZGVRHSvzbKZ
CMhpB1UcMPNT4eHh7zPzSmvprfnAttNkuJ+ZG8Im9G9WynQhotstUweEEPdLKefZ2PeQlPI8AKMs43
lmpON5u/
JMIloIGxVM09lJRHDC7lpN054282ysKpgQQqTY7Wfme4wEEFQOI4S4zTCMcXb2eb3eIpfLZaemqlCc
MagjD0W1Rtd1AoATJ07MF0Lcq2na2T6fb7P1Omb2E1Ezy9gxIUSylHKRTej6GQBLvV7vRlgiDQjILh
+GfdOxbwFcablPiaZpvaSUowBYO3oCwFsc6D1yyuFwZv6YiAoBjLZM5SLQqt1O5nkZEWUyc7LN3Bsu
l+vFnJycjXYVC1YHw+Rl0wlaaWPfBACRCJSWlh8/aWpXzLM+yJn5dQC/
WKtUzJyNnoZh3G0XBSGiNGZ229h3p5SyIyyRDmbepWlagt/
vX2Uec1ltn8bMbpufiZlSyqttkkcB4G4p5a12zqLf7z/q9XrDbNYoFGcUKkKhqNaUOhRZWVk/
6roe5vV6V8Mm0dGaoc+BPhKJzJxiU9K3KCYm5lkp5SpYIg3MXALgC9g4E8y8BRZnwrx3LDNfCXvp5/
eJaBICLa5PCWbeCWALgp0JAHjO/
GNds5WI1to5E8y82uVy3Z6Tk7MYwCWnaMMSZp5dydHIm5qmbQfwinVC07QBzDwNwVLXG4QQc+ycCSL
qQUTXlkafrPdi5lgb+54momJYvnMOyIQPlFJm2EVczMhWLAV3oX2XiBxkI/
ltfp7GRGRnx6HMzMyVRKQcCsUZj3IoFNUar9dLAKBpmt/
j8aywe7u2g4jcpnKmtWvn5piYGHdOTs4SIupks/Qz2Msuf1NJguMYZo7g3/
QwyrNRCDFaSrnc5gFW+mZuHTtCRIuIaLzN3FOwRARMfiaiOeaDz8oOl8t1c3Z29muwJKRWBjNvEUI8
```

```
AGCVTeRisxBiTiUKo4OllPcgWNNivxBiAjMHiXsBGCCEqAdgks3cKmbugmAH8lVd11eYEY/y9y/
Sdb2/z+ebAZtupAgk2F5i/
Rli5q0AvmJmW5VOIcQO2B9VZTudzhs9Hk8RALs29ArFGYVyKBTVGiGEBgCtW7ceYi33/
B3GAHBb31CZebemaTfl5OSkAOhpXcTMX1SS/7CHmYMS7ph5BhF9R0R2b/
HbnE5nrJRyKRE1sllbYFORUCSEmIVAW3Hr9S+Zb/
DWh2s2gKkA7M7vf3U6nd2zs7OTEJyzYAsz7w4PDy99w7c6bz8T0aOGYdiV6t6FQOVKBYEvDjRze8zs
sWK1vZ+u69mGYcy32W8TMzeziTzNBfCCYRhBxzBENNjv9z9gd2yCQIKtZvMztB+BvBO77/
xLZrbVwjDn/
T6fbxgCuTaYPXu2w+46heJMQTkUiuqOAwAiIyPtWm0HQURTAVwBe52CHoZhTIBNngMz/
wR7Jc4jCJRmWhP7lhNRupTyPZs1Pzgcjl4ej+cN2LQINxMgrZLfIKJpZhTCyptE1BWBZEsrTzDzZJt
7nHQ4HD19Pt+VRBQ0XwnZQogeHo9nKhFZG3HlCiGSpJT/
tfkuJpk5LPfb7PkkgJcRXHY7lIh2GIax0sax2oWAsFYry5rFLpdrDID3EBx5uhNAr0qOJX4C8D2CtT
AKADwGwC66tFfX9buJKEgRtBQKCHg91rhx4+sBwO/
3q2MPxRmNcigU1ZqaNWtGAoCun1J+8rtSykJYHAZmLjaTJuNgaeplzh8D0MimTDAfgbdP68Pra13XH
2LmD20ervvCw8Ov93q9z1Xypvx7CZBBVRvM/9/e2cfGcZxn/
Hlndo88HkXe7fks6sMMZYmQIysNhBaO4gipHcgyYpEJQtjyR9KmkuNYQFsoif2HHdit69i1m6SF08a
RWjuSkMSOgjaJ7NhRbCmVjVYSohpObFigG4v6pD5Ic/d4JE8id2/
n7R+7J9C7c7KFtpAAzg8gDpzd2Z29IzjvzTzv8/
IuZi5BE5ggcvT8BqVdKyGE6FFKtSqlkgXEtMRBzkoAXwBwm+aUPw3DcGOD1Mxf6dJDAfwzEd2HtAvm
/QB+qZR6Een39jAAiXQWzfOO49xaLpefQkIHEgeRRWg+W0R1W/
4V0yqsTuv3ZWb+O837N0JEfWEYbtGMvZ4BdI5cLvcRALAsS+eIajBcMpiAwjCjqVarAAClVMqYaDoc
2Se/
TEQpa2UhxG1hGK7UHYvTQHO6SZmIysntCmY+YVnWHUEQbE9Orsx8Wgix0vf9uxoIDBuxDZHPQ1Jn8R
YReQ2Egt9g5q8iUV015nYiOhaG4Q6NIFWLlLIPwII4e+M9UOQ0+VeUNtZ62bKsR5VSOm3EXgA3I+2q
udlxnMcRiSCT7pRDiFwrk/f5teM4n/
U87xGka6y8rJTqR5S1k4KZn4RGd0JE65h5g0ZPcRbAFzmqypoMahBn67xn5aSeLSKEMAGF4ZLGBBSG
mY4NAPv37/8aoqqUVc05hwH8PTS22UT0FQALAPxtg+tL3YpB/
C00aWhVBXBzrVbbrFmOd23bvp6ZP9lg26IR+5h5BdI+GKeZ+dcA1iQ7ENHTRHR7Ul8Qc78Q4qUwDHc
g7eKpJQ4YhnSrGUT0oFLqDoqqek4f3/7m5ua1QRA8p9m+eRfRZJwMJl4qFot3uq67GcB1iT4VRKtBC
xPtrzqO8xnXde9EtD0xnQEhxLNE9FSDR/
s2gKS1OJj5YaXUH+tEtkKI+xBZbv9Bos8EET1IRH3JPuVyeScAKKXMlofhksYEFIYZjVLKAgDHccrM
fJ1m8h8lonuQsHeO+S4z+8z8RKPrJ7c5Yg5qAgYIIe4goq8j0micI94uuEEpdUUsGvygHEO01ZL8lj
wphPge6Y2rXmDmj0FfB+Qpx3Eej4WgqfHriCfJ55n5eY2WYQszL6GEFXesEVk9OTm5NbmaEFNCWoD5
huM4N5fL5QeIaK2mT6gRYP5XoVBY7XneJ4no6cSxs0T0HWbe2uDRfghgjWZL6lkhhEtpMy4g2nZ6GG
mtSkVKuVa3JQXgn4aHh/fHYzIBheGSxgQUhplOOwBce+213yOia5IHhRBfijMgUqJJAP3QF/
tqCEf1HVKTNUWOkT3QZzHcwMxhGIbbL+A+kwCySPs1gIj+Ril1n6bPawDmQmPrzcwvOY7zZc/
znkFakNpoDJsLhcIjSqntmnHsFkJ4SG8xnACwKgiCBxppRDQcz2QyN46OjvbptlRikis0v7dt+9Plc
nkhNAXgiOgflVK6VFMA2I1Ic5J8pr1CiO9zVEMkySuIsoOSlVxPSSlvVEo9hIR7JjPvtCzr3GpUg+D
UYLhkMAGFYUajlCoBgJTyxuQxIlqnlHoA6QqcrzHzq9CnUp4PD5ptgvhb8CxohH1Syl5ENUR26nQYj
Ygnn6QgEUT0EIC7NNc6FL+m/
BWY+bXYa+JR6J06dbxQLBbv9DxvE9L1Ow4A2MHM9yTaPdu2VwohVjZYPUnBzBMAbgqCYHnSN+I8fU4
LIVZJKTOsL5X+C0TvkU4f8hYAQjpj56Bt219SSm3T3K+K9BZMPVj8WBiGX0VaTzEA4N+CIDhy+eWXf
xQwAYXh0scEFIYZTVdX1zogXb4cUXnrPqSdH48S0b9fQKokgGjiY2ZbMym8KoT4j3gpPElvGIavEdG
OOIXwf8tGZv4jpItMjQAYT+oYYo41NTX1jo6OroHe9CoFM/
+mUCisiXUJdyeOnUbkgPnNRHtVSvnpIAicRr4MOqSUfUKIOcz8QVdvKgBW5vP5kampqR0aUewgM18D
TfYFIu3GANLBgUdEt/
i+vxX6IE6XdQMAfyaEuAGJYm7xFtfXAPwDEbUWi8UeACiVSjpNi8FwyWACCsOMxXXde6WUnwJSAcVG
RILJnumNzDzGzLsRpVOmYObBRvciolAjLjwCYBOnC5EBwO2O47wA4OcAlr7Po7wvcQrqENLPNAHgv9
HAzwJATxAEH/6gkzwz/
96yrJsqlcoKjS7hDIAHdb4WUsrP2bZ9Apqy4o0gorVhGFYvZCsIwOpisXggtgnXpcpeRpr6HDE7AXw
22SiEuJUjE7Lkltmh5LnT+HYmk3lLKZWyFwewlpkfSgYipVLpE+e5nsFw0THFwQwzFiL6fDabhW2/
x4DwF8x8iIi+lTxfStkXhqFWP8DMp4loNUeVNpMOkEeRzug4Q0R/wczbKF1Ear3jONs8z/
sRgE9d8IOlx9Yfm0a9mbyXlPJzYRhOEFGqIBoR3WJZ1lStVvu55rIu0lkWw7G519w4/
TF5vduYeQBAFVHhr3q/W/P5/
E7XdfefZzJP8rhSaj8RHdA87xYAHyeiqxLta4rF4h7P856ExiacmbdSXK49DEMopcDMUEohCIIfCiH
+RCkFpRQAQCkF13UfbmlpedSyrGumn+/7/
tE33njjke7u7nXZbHZFvR8z48yZM4eOHz+Ozs7OvQCy9T5KKYyNjf2WmR/
LZrMLwzAEM6OtLZLvHDhw4LypzQbDxSb1n8xgmCm4rtt2+PDh37z55ptXdXd3IwiCE67rvtLR0fH56
ZMhEcF13e0nT57c39LSMmvRokX319sBQClVGxwc/
K4QIjNv3rw7hRDn1PgjIyPPVCqVwwsWLLhLSjm73mdoaOjpSqVyaNGiRRukllPr1xsdHd3+zjvv/
LKjo2NxZ2fnPcyM6WNh5mqlUnkxn8+vmX5MKTURBMFQJpNZyMz1c8HM1YGBgW+Oj497+Xx+fmdn55/
X/ShGRkZ+NDg4+FsA6Orquj6fz/fU+3qe99zx48f3dXR0/
```

```
GGpVHqPO2QQBO8eOnToX+bOnXtTLpdbRkQIw7B28uTJzZVK5XR3d/
c6KeX8+sTLzCiXy3tc1+1nZqupqSlfKpVWEVHL2NjYgbGxsQFmttrb269ubm7+UDxuKKWmRkdHD8ya
NWuxlDJXbw+CYKxarR4FkGltbe0ioqb6sVqtdnZqaupdIrIymcxsIpJxezUMwykAgoiapZTNAFCr1f
5v/phiLMsCESEIAhfAlJQyk81mLyOi+iqYPz4+/
rtcLjevpaVlnhAC9Z+pqamBarX6dkdHx+rp7dVqdfDYsWPzmfm29evX/
+R9hmAwXDRMQGGY0WzdunV1a2vr02NjY6dqtdpZZlZtbW1X2bZ9GRBNhr7vD42Pjx8mImJmamlpmZ3
NZj9Un3yr1eqA7/
sVZiYpZba1tfVKIspMTU0NT05ODjEzEZHI5XKdAGQQBBXf98sAiIislpaWTiLK+L4/6vv+CABiZrJt
u7WpqenyeLKsEZGcnJwcDsPQl1I2ZTKZy4gomqV832Vm2LbtxONE3D7KzLV4DEREQkrZppSaUkr58T
PWj9lSymalVKCUCqb3YebMRfh4DAAcx7nJ87xnmXnD+vXrf3Cxx2MwNMIEFAYDgI0bN95FRLOJqAYg
7O7u7svlcssnJiZ2Hjx48KfxpBwSUU0IUbviiisWz5s3795yufyD/
v7+F4UQoVKqxsxhsVjMLVy4sO/1119/Ku6j6q+O47SMjIyM1tuEEKqrq6v1yiuv/
MKuXbueUEqFQggVhmFYq9XCnp6ej+Zyuceam5s/
w8ycz+fL9TGXy+UupdRXiOgdx3GeBADXdZcA2EVEc4joi4dO9D57nfcJxnD3Jdt
d1lxDRBiL6caFQeCXRp91xnMr0NmZu9TzvZ0Q0p1AoXENRKfBzbNmypTl+BsrlcpTNZgkAxsfHadWq
VR8RQji7du36TwDIZDJ09uxZ6u3t/cvh4eFoBHSqAAABP0lEQVTt+/
btO2HbNtm2TQAwZ86c3OLFi2/
p7+9/7siRI2OWZZFlWeT7PnV1dbUODg5WfD8ny7JISkkAEAQBLV26dL7v6JarU6cOnWqIqIWqQWkWq1GA
CCEoBUrVmzo7//vud5ZwEgDENavnx5b3Nz8/W7d++
+VwhBQghaunTpbW7u7v/vud5ZwEgDENavnx5b3Nz8/W7d++
ntbW1t22bJlTA4RhOLpnz56vT+/T3t4+6+qr/7ro0ePfqe3t/
eZTZs2fZyIjtx9992nGv4RGwwGg+HSpExEgDENavnx5b3Nz8/W7d++
c+x5MFqS4JXNedXy6XP1Da5f831Wp1RmQmlMvlvOd5j11oP2ZOFXEzGAwGg8FgMBgMBoPBoPBoPBYPBoPBoPBoPBYPBoPBaDwI/
wCN2zKfNC38HwAAAABJRU5ErkJggg==" alt="Diagram" /></div>

Each arrow represents an operation that contributes to the overall trace,
identified by the common `Trace ID: A`.

**⚡ Real-world insight:** OpenTelemetry (OTel) is an industry-standard set of
APIs, SDKs, and tools designed to standardize the collection of telemetry data
(logs, metrics, and traces). It's vendor-agnostic, meaning you can instrument
your application once and export the data to various tracing backends like
Jaeger, Zipkin, or commercial solutions like Datadog, New Relic, or Grafana
Tempo. As of 2026-05-15, OpenTelemetry is the recommended approach for
instrumenting applications for distributed tracing.

**AI/Agent Context:** Distributed tracing is incredibly powerful for AI agent
workflows, which are inherently distributed and often involve multiple
sequential or parallel steps:
- **Agent Orchestration:** Trace the entire lifecycle of an agent's task, from
initial prompt to final output.
- **Tool Chaining:** Visualize the sequence of tool calls an agent makes,
including their individual latencies.
- **Multi-Agent Coordination:** If multiple agents collaborate, traces can
show how requests flow between them, revealing bottlenecks in communication or
decision-making.
- **External Service Dependencies:** Pinpoint which external APIs (e.g.,
vector databases, knowledge bases, LLM providers) are slowing down the agent's
response.
This allows you to optimize agent prompts, improve tool reliability, and
understand complex interactions.

## Implementing an Observability Strategy

Building an effective observability strategy isn't just about installing
tools; it's about integrating telemetry collection into your development
lifecycle from the start.

1.  **Instrumentation:** This is the act of adding code to your applications
to generate logs, metrics, and traces.
    - **Automated Instrumentation:** Many frameworks and libraries offer
```

```
automatic instrumentation (e.g., for HTTP requests, database calls) through
agents or SDKs. This is a great starting point.
    - **Manual Instrumentation:** For business-specific logic, critical
decisions, or complex AI agent steps, you'll need to manually add calls to
your chosen observability library (like OpenTelemetry SDKs).
    - **Key Idea:** Use OpenTelemetry (OTel) for instrumentation. It provides
a unified standard for collecting all three types of telemetry. As of 2026,
OTel has stable APIs and SDKs across many languages and is widely adopted.

2.  **Contextualization:** Ensure your telemetry data is rich with context.
    - **Logs:** Include `request_id`, `user_id`, `session_id`, `service_name`,
`version`, and any other relevant business identifiers in your structured
logs.
    - **Metrics:** Add meaningful labels (tags) to your metrics, such as
`service_name`, `endpoint`, `status_code`, `method`.
    - **Traces:** Ensure trace and span IDs are correctly propagated across
service boundaries. Add relevant attributes (tags) to your spans for filtering
and analysis.

3.  **Collection & Storage:**
    - **Logs:** Use log agents (e.g., Fluentd, Vector) to collect logs from
application instances and send them to a centralized logging platform (e.g.,
Grafana Loki, Elasticsearch).
    - **Metrics:** Use a pull-based system like Prometheus to scrape metrics
endpoints, or a push-based system to send metrics to a time-series database.
    - **Traces:** OpenTelemetry Collectors can receive trace data from your
applications and export it to various tracing backends (e.g., Jaeger, Grafana
Tempo).

4.  **Analysis & Visualization:**
    - **Dashboards:** Create dashboards (e.g., in Grafana) to visualize key
metrics and log trends.
    - **Alerting:** Set up alerts on critical metrics (e.g., error rates,
latency spikes) or specific log patterns.
    - **Trace Analysis:** Use tracing UIs to explore individual traces,
identify bottlenecks, and understand service dependencies.

5.  **Feedback Loop:** Observability is not a one-time setup. Continuously
refine your instrumentation, adjust your alerts, and improve your dashboards
based on new insights and evolving system behavior.

## Mini-Challenge: Designing Observability for an AI Agent Task

Let's apply these concepts to a real-world scenario involving an AI agent.

**Challenge:**
Imagine you are building a system where an AI agent's primary task is to
**research a complex topic, synthesize information from multiple sources (web
search, internal knowledge base), and generate a concise report.** This agent
uses several tools internally.

Propose what specific logs, metrics, and distributed traces you would
implement to ensure this AI agent's workflow is fully observable. Think about
what information would be most valuable for debugging, performance analysis,
and understanding the agent's decision-making.

**Hint:**
Consider each stage of the agent's process: receiving the request, planning,
tool usage, information synthesis, and report generation. What are the key
data points at each stage? How would you connect these points across different
services or even within the agent's internal "thought process"?
```

```
<div class="hint-box"><div class="hint-label">💡 Click for a possible solution
hint!</div><div class="hint-body"><p>
Consider:
<br>
<b>Logs:</b> Agent's intermediate thoughts, prompt used for an LLM call, raw
output from web search, specific knowledge base articles retrieved, decisions
to retry a tool.
<br>
<b>Metrics:</b> Latency of the overall task, latency of individual tool calls
(web search, KB lookup, LLM inference), number of tokens consumed, success/
failure rate of report generation.
<br>
<b>Traces:</b> An end-to-end trace for each research request, with spans for
each planning step, each tool invocation, and the final synthesis step. Ensure
trace IDs propagate to any external services (like web search API or LLM
provider).
</p></div></div>



## Common Pitfalls & Troubleshooting

Even with the best intentions, implementing observability can present
challenges. Being aware of these common pitfalls can save you a lot of
headaches.

- **Logging Too Much or Too Little:**
    - **Too Much:** Excessive logging can overwhelm your logging system, incur
high costs, and make it difficult to find relevant information. Avoid logging
verbose data that can be derived from other metrics or traces unless
absolutely necessary for debugging.
    - **Too Little:** Insufficient logging means you lack the critical context
needed to diagnose issues. Balance verbosity with relevance. Use appropriate
log levels (DEBUG, INFO, WARN, ERROR) to control what gets emitted in
different environments.
    - **What to observe/learn:** Focus logs on state changes, critical
decisions, and errors.

- **Lack of Context in Telemetry:**
    - Logs, metrics, or traces without proper tags, labels, or attributes are
far less useful. If a log doesn't include a `request_id`, it's hard to
correlate it with other logs from the same request. If a metric doesn't have a
`service_name`, you don't know which service it belongs to.
    - **What can go wrong:** Debugging becomes a nightmare, as you can't
filter or group related data points.
    - **Optimization / Pro tip:** Standardize your attribute names across all
services and telemetry types. OpenTelemetry provides conventions for this.

- **Ignoring Trace Propagation:**
    - For distributed tracing to work, trace context (trace ID, parent span
ID) *must* be propagated across all service calls. If a service fails to
propagate these headers, the trace will be "broken," showing only a partial
view of the request's journey.
    - **What can go wrong:** You lose the ability to see the full end-to-end
flow, making it impossible to pinpoint cross-service bottlenecks.
    - **Troubleshooting:** Verify that all inter-service communication
libraries (HTTP clients, message queue producers) are configured to propagate
OpenTelemetry trace context headers.

- **Observability as an Afterthought:**
```

```
    - Trying to bolt on observability to an existing, complex system after
it's already experiencing issues is incredibly difficult and costly.
    - **What can go wrong:** You'll spend more time firefighting and less time
innovating.
    - **Pro tip:** Treat observability as a first-class concern in your system
design. Integrate instrumentation into your development workflows and CI/CD
pipelines.

- **Alert Fatigue:**
    - Setting up too many alerts, especially on non-critical metrics or
without proper thresholds, leads to "alert fatigue." Engineers start ignoring
alerts because most of them are false positives or unimportant.
    - **What can go wrong:** Critical alerts get missed amidst the noise,
leading to extended outages.
    - **Optimization / Pro tip:** Focus alerts on actionable events that
indicate a real problem affecting users or business operations. Use "the four
golden signals" of monitoring: latency, traffic, errors, and saturation.
Review and refine your alerts regularly.

## Summary

In this chapter, we've explored the foundational concepts of observability,
which is paramount for understanding and managing complex distributed systems,
especially those incorporating intelligent AI agents.

Here are the key takeaways:

- **Observability vs. Monitoring:** Observability is about understanding *why*
your system behaves a certain way by exploring its internal state through
external outputs, while monitoring is about *what* is happening based on
predefined metrics.
- **The Three Pillars:**
    - **Logs:** Structured event records for debugging, auditing, and
understanding specific sequences of events.
    - **Metrics:** Numerical measurements over time for monitoring trends,
alerting, and capacity planning.
    - **Distributed Tracing:** End-to-end visualization of a request's journey
across services for performance bottleneck identification.
- **AI/Agent Workflows:** Each pillar provides unique insights into agent
decision-making, tool usage, and performance, crucial for reliability and
optimization.
- **OpenTelemetry:** The unified, vendor-agnostic standard for instrumenting
applications to collect all three types of telemetry data.
- **Strategic Implementation:** Observability must be integrated early, with
rich context, robust collection, and continuous analysis.
- **Common Pitfalls:** Avoid logging imbalances, lack of context, broken
traces, treating observability as an afterthought, and alert fatigue.

By embracing these timeless engineering principles, you'll gain unparalleled
insight into your systems, enabling you to build more resilient, performant,
and reliable applications, no matter how complex they become.

## References

- [OpenTelemetry Documentation](https://opentelemetry.io/docs/)
- [Microservices Architecture Style - Azure Architecture Center](https://
learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/
microservices)
- [Google Cloud - The Four Golden Signals of Monitoring](https://
cloud.google.com/blog/products/operations/sre-fundamentals-monitoring-
production-systems)
- [Prometheus Documentation](https://prometheus.io/docs/introduction/
```

```
overview/)
- [Grafana Loki Documentation](https://grafana.com/docs/loki/latest/)

This page is AI-assisted and reviewed. It references official documentation
and recognized resources where relevant.
```

CHAPTER 10

Infrastructure Automation and Deployment Strategies

Imagine you've just built an amazing new feature for your distributed system—perhaps an intelligent agent that personalizes user experiences. Now, how do you get it from your development machine into the hands of millions of users without causing chaos or downtime? Manually configuring servers, networks, and databases across multiple environments is not just tedious; it's a recipe for inconsistent setups, human error, and sleepless nights.

This is where infrastructure automation and sophisticated deployment strategies become your best friends. In modern systems engineering, especially with the dynamism of AI and agentic workflows, the ability to rapidly and reliably deploy changes is paramount. This chapter will guide you through the timeless principles and practical approaches to automate your infrastructure and deploy your applications with confidence and control.

By the end of this chapter, you'll understand why automation isn't just a luxury but a necessity, how to define your infrastructure as code, explore various deployment strategies to minimize risk, and see how continuous delivery pipelines ensure your innovations reach production smoothly. We'll build upon our understanding of distributed systems to ensure these practices support scalability, resilience, and observability.

The Foundation: Why Automate Infrastructure?

In the early days of computing, setting up a server meant physically installing hardware, cabling, and manually configuring operating systems and applications. Even in virtualized environments, manually clicking through a cloud provider's console to provision resources is slow, error-prone, and doesn't scale.

The Problem with Manual Provisioning

Manual processes lead to:

- **Inconsistency:** Different environments (development, staging, production) can diverge, leading to "works on my machine" issues.
- **Slow Deployment:** Provisioning new environments or scaling up takes significant time and effort.

- **Human Error:** Typos, missed steps, or incorrect configurations are inevitable.
- **Lack of Auditability:** It's hard to track who changed what and when.
- **Scaling Challenges:** Replicating manual steps for dozens or hundreds of servers is impractical.

The Power of Automation

Infrastructure automation addresses these issues by treating infrastructure as a programmable resource. It's about defining, provisioning, and managing computing resources through code and automated processes.

Why does this matter for distributed systems? Distributed systems are inherently complex, with many interconnected services. Automation ensures that each service's dependencies are correctly provisioned, that scaling events are handled reliably, and that new environments can be spun up quickly for testing or disaster recovery. For AI agents, which might require specific GPU instances or large data storage, automation ensures these specialized resources are available precisely when and where they're needed.

Infrastructure as Code (IaC)

At the heart of infrastructure automation is the concept of **Infrastructure as Code (IaC)**.

What is IaC?

IaC is the practice of managing and provisioning infrastructure through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools. It applies software engineering practices—like version control, testing, and continuous integration—to your infrastructure.

Think of it like writing a recipe for your entire infrastructure. Instead of manually cooking each dish, you write down the ingredients and steps, and a machine follows them precisely every time.

📌 **Key Idea:** Your infrastructure becomes a versioned, testable artifact, just like your application code.

How IaC Works

IaC tools allow you to define the desired state of your infrastructure (e.g., "I need two virtual machines, a database, and a load balancer") using a declarative language, often YAML or HCL (HashiCorp Configuration Language). The tool then takes this definition and makes the necessary API calls to your cloud provider (AWS, Azure, Google Cloud) or virtualization platform to achieve that state.

⚡ **Real-world insight:** Popular IaC tools include Terraform (cloud-agnostic), AWS CloudFormation, Azure Resource Manager, and Pulumi. While the tools vary, the underlying principle of declarative infrastructure remains the same. As of 2026, these tools continue to be industry standards.

Benefits of IaC

- **Consistency:** Eliminates configuration drift between environments.
- **Repeatability:** You can recreate your entire infrastructure on demand, perfect for testing, staging, and disaster recovery.
- **Speed:** Provisioning resources takes minutes, not hours or days.
- **Version Control:** Infrastructure definitions are stored in Git, providing a complete history of changes, who made them, and why. This is invaluable for auditing and rollbacks.
- **Idempotency:** Running the IaC script multiple times yields the same result, without unintended side effects. If a resource already exists and matches the desired state, the tool does nothing.

Step-by-Step Implementation: Building an IaC Definition

Let's incrementally build a conceptual IaC definition to provision a simple setup for an AI agent. We'll use a generic YAML structure, which is common across many IaC tools.

Step 1: Define the Resource Group

First, we need a logical container for our resources. This helps organize and manage them.

```yaml
# iac_config.yaml - Initial file
resource_group:
  name: "ai-agent-production"
  location: "eastus"
```

Explanation:

- We're declaring a `resource_group` block.
- `name`: Assigns a unique name, `ai-agent-production`, to our group.
- `location`: Specifies the geographical region, `eastus`, where these resources will reside.

Step 2: Add a Virtual Machine for the AI Agent

Next, let's define a virtual machine that will run our AI agent's core logic.

```
# iac_config.yaml - Adding the VM
resource_group:
  name: "ai-agent-production"
  location: "eastus"

virtual_machine:
  name: "ai-agent-compute-01"
  resource_group_name: "ai-agent-production" # Reference the resource group
  size: "Standard_D8s_v3" # A compute-optimized size for AI tasks
  image: "UbuntuServer:20.04-LTS:latest" # Using a recent Ubuntu LTS
  network_interface:
    name: "ai-agent-nic-01"
    private_ip: "10.0.0.4"
    public_ip: "none" # For security, often no public IP directly on compute
```

Explanation:

- We've added a `virtual_machine` block.
- `name`: Identifies this specific VM.
- `resource_group_name`: Links this VM to our previously defined `ai-agent-production` group.
- `size`: Specifies the VM's hardware profile (CPU, RAM). `Standard_D8s_v3` is a common choice for moderate compute needs.
- `image`: Defines the operating system and version to be installed. We're using Ubuntu Server 20.04 LTS, a stable and widely used OS as of 2026.
- `network_interface`: Configures how the VM connects to the network, assigning a private IP and explicitly not assigning a public IP for enhanced security.

Step 3: Include a Database for Agent State

AI agents often need to store state, user profiles, or model outputs. Let's add a managed database.

```yaml
# iac_config.yaml - Adding the Database
resource_group:
  name: "ai-agent-production"
  location: "eastus"

virtual_machine:
  name: "ai-agent-compute-01"
  resource_group_name: "ai-agent-production"
  size: "Standard_D8s_v3"
  image: "UbuntuServer:20.04-LTS:latest"
  network_interface:
    name: "ai-agent-nic-01"
    private_ip: "10.0.0.4"
    public_ip: "none"

database:
  name: "agent-state-db"
  resource_group_name: "ai-agent-production"
  type: "PostgreSQL"
  version: "15" # Latest stable PostgreSQL version as of 2026
  sku: "GeneralPurpose_4_vCPU" # A general-purpose SKU
  storage_gb: 250
  backup_retention_days: 7
```

Explanation:

- A `database` block is added.
- `name`: Unique identifier for the database.
- `resource_group_name`: Associates it with our resource group.
- `type` and `version`: Specifies the database engine (PostgreSQL) and its version (15, which is a recent stable release).
- `sku`: Defines the performance tier and resources allocated to the database.
- `storage_gb`: Sets the initial storage capacity.
- `backup_retention_days`: Configures automatic backups, a critical operational consideration.

This incremental approach demonstrates how you build up complex infrastructure definitions piece by piece, with each addition explained and justified. When an IaC tool processes this `iac_config.yaml` file, it will create or update these resources in your cloud environment to match this desired state.

Deployment Strategies

Once your infrastructure is automated, the next challenge is how to safely deploy your application code. Simply shutting down the old version and starting the new one can lead to downtime, which is unacceptable for critical systems. Modern deployment strategies aim to minimize risk and downtime.

1. Rolling Updates

What it is: This is the most common deployment strategy, especially in container orchestration platforms like Kubernetes. New versions of your application are gradually rolled out, replacing old instances one by one or in small batches.

How it works:

1. A few instances of the old version are terminated or drained of traffic.
2. New instances of the new version are started and brought online.
3. Traffic is directed to the new instances.
4. This process repeats until all old instances are replaced.

Why it exists: Minimizes downtime and allows for gradual resource consumption. If issues arise with a new version, the rollout can be paused or rolled back, affecting only a subset of users.

⚠️ **What can go wrong:** During the transition, both old and new versions of the application are running simultaneously. This requires backward and forward compatibility for APIs and data schemas. If the new version has a critical bug, it can still affect users as it rolls out.

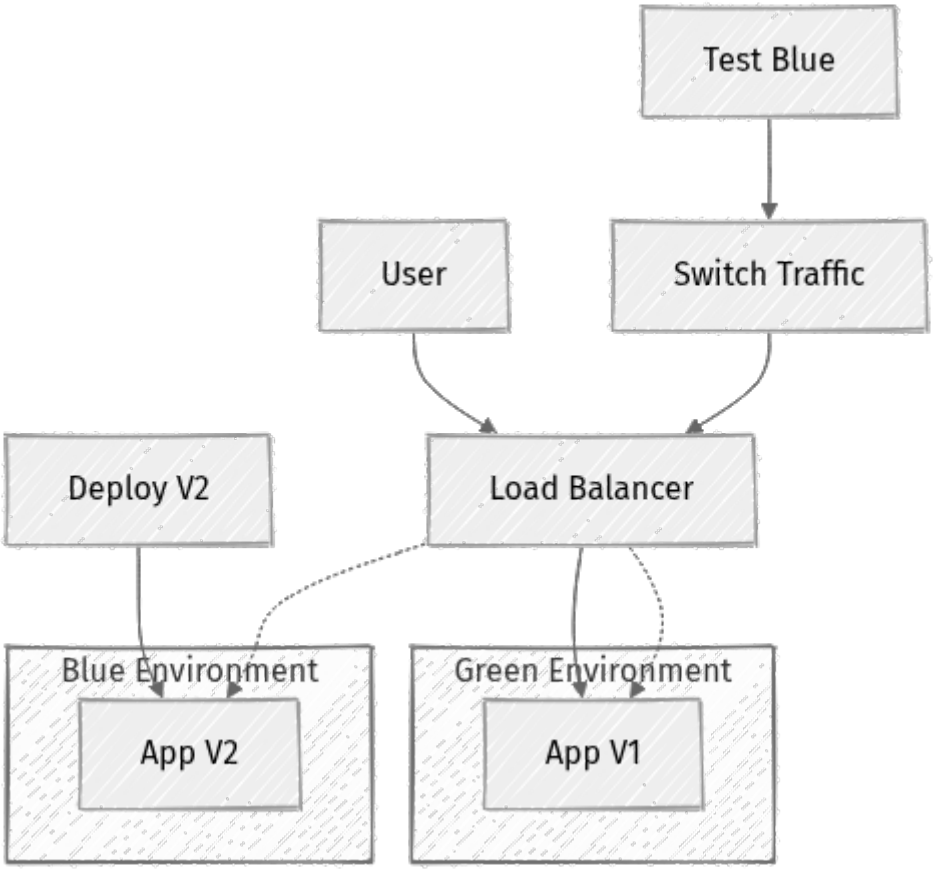
2. Blue/Green Deployments

What it is: You maintain two identical production environments, let's call them "Blue" and "Green." At any given time, only one environment is live and serving user traffic.

How it works:

1. **Green (Live):** Currently serving all traffic (e.g., App V1).
2. **Blue (Idle):** Deploy the new version of your application (e.g., App V2) to this environment.
3. **Testing:** Thoroughly test the new version in the Blue environment.

4. **Traffic Switch:** Once confident, switch your load balancer or DNS to direct all incoming traffic to the Blue environment. Green becomes the idle environment.
5. **Rollback:** If issues appear in Blue, you can instantly switch traffic back to the Green environment.



Why it exists: Provides instant rollback capabilities and zero-downtime deployments. You have a completely isolated environment to test before going live.

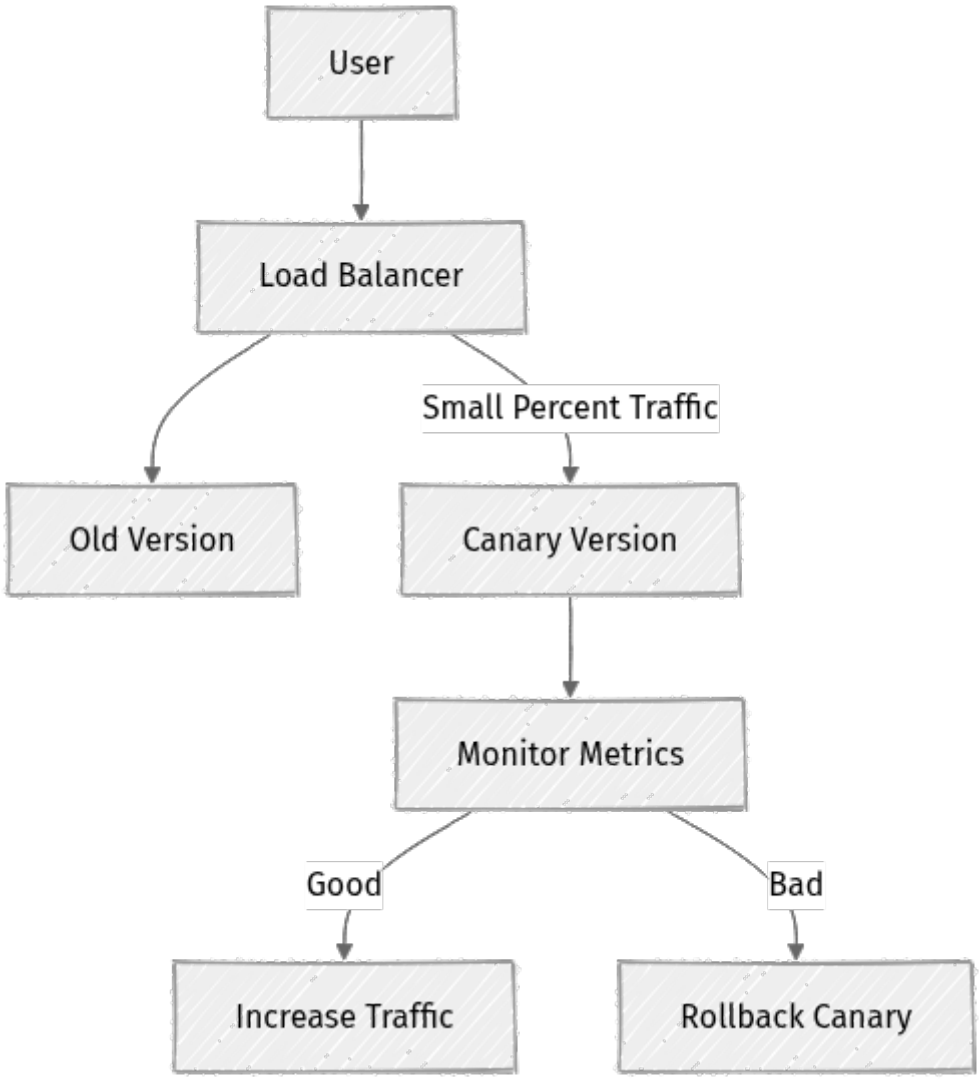
⚠️ **What can go wrong:** Requires double the infrastructure resources, which can be costly. Database migrations can be tricky; ensuring both blue and green environments can work with the same database, or managing database changes for each switch, adds complexity.

3. Canary Deployments

What it is: A phased rollout where a new version is introduced to a small subset of real users (the "canaries") before a full rollout. This allows you to monitor its performance and stability in a live environment.

How it works:

1. **Deploy Canary:** A small percentage (e.g., 1-5%) of user traffic is routed to the new version.
2. **Monitor:** Closely monitor metrics (errors, latency, CPU usage) for the canary group.
3. **Gradual Increase:** If the canary performs well, gradually increase the percentage of traffic directed to the new version.
4. **Full Rollout/Rollback:** If all goes well, roll out to 100%. If issues arise, immediately roll back the canary traffic.



Why it exists: Reduces the blast radius of potential issues. You get real-world feedback on the new version before it impacts your entire user base. Ideal for AI agent updates where subtle behavioral changes might only be apparent in production.

⚠️ **What can go wrong:** Requires sophisticated monitoring and alerting to detect issues quickly. The small canary group might not expose all problems, especially those related to scale or specific user patterns.

A/B Testing vs. Deployment Strategies

⚡ **Quick Note:** A/B testing is often confused with deployment strategies. While both involve routing traffic to different versions, A/B testing is primarily about **feature validation** (which version performs better against a business metric), while deployment strategies are about **safely delivering a new version** of the software. You can run an A/B test within a canary deployment.

Continuous Integration and Continuous Delivery (CI/CD)

Automating infrastructure and choosing smart deployment strategies are components of a larger system: CI/CD.

Continuous Integration (CI)

What it is: The practice of frequently integrating code changes from multiple developers into a central repository, followed by automated builds and tests.

How it works:

1. Developers commit code frequently (multiple times a day).
2. Each commit triggers an automated build process.
3. Automated tests (unit, integration) run against the new code.
4. If tests pass, the code is integrated. If they fail, developers are immediately notified.

Why it exists: Catches integration issues early, reduces the time developers spend debugging merge conflicts, and ensures the codebase is always in a releasable state.

Continuous Delivery (CD)

What it is: An extension of CI where code changes are automatically built, tested, and prepared for release to production. It ensures that you can release new changes to customers rapidly and sustainably.

How it works:

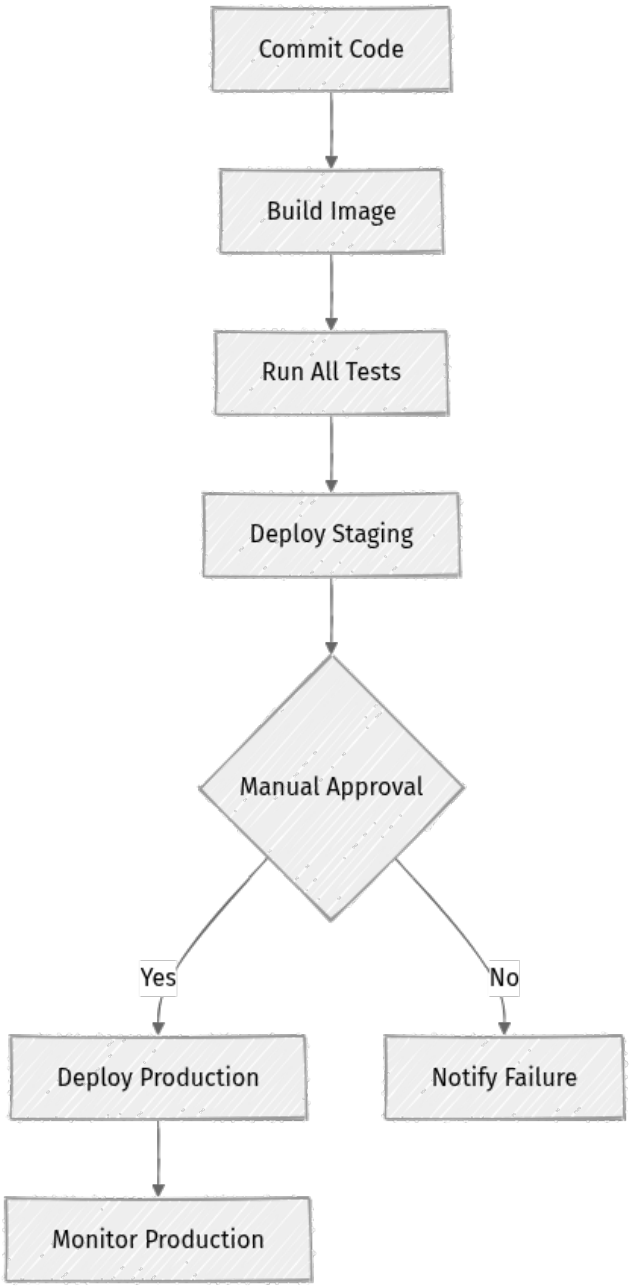
1. After CI, the build artifact (e.g., a container image, an executable) is stored.

2. Automated tests (end-to-end, performance, security scans) run against this artifact in a staging environment.
3. If all tests pass, the application is ready for deployment. The deployment itself might be manual (a button click) or fully automated, depending on the organization's maturity and risk tolerance.

🧠 **Important:** Continuous Deployment is a further step where every change that passes all automated tests is automatically deployed to production, with no human intervention. Continuous Delivery means it can be deployed, not that it is automatically deployed.

The CI/CD Pipeline

The CI/CD pipeline is a series of automated steps that your code goes through from development to production.



Explanation:

- **Commit Code:** The starting point for any change.
- **Build Artifact/Image:** Compiles code, creates a container image or other deployable artifact.
- **Run All Tests:** Executes various automated tests (unit, integration, static analysis).
- **Deploy Staging:** Uses IaC and a deployment strategy to deploy to a non-production environment for further testing.
- **Manual Approval:** A common gate for critical production deployments, ensuring human oversight.

- **Deploy Production:** Applies a chosen deployment strategy (e.g., Canary, Blue/Green) to the live environment.
- **Monitor Production:** Essential feedback loop to ensure the deployed changes are performing as expected.

Why CI/CD is crucial for AI Agents: AI models and agent logic are constantly evolving. A robust CI/CD pipeline allows for rapid iteration, testing new model versions, deploying agents with updated decision-making algorithms, and quickly rolling back if an agent's behavior deviates from expectations. This agility is critical for competitive AI systems.

GitOps: The Evolution of CD

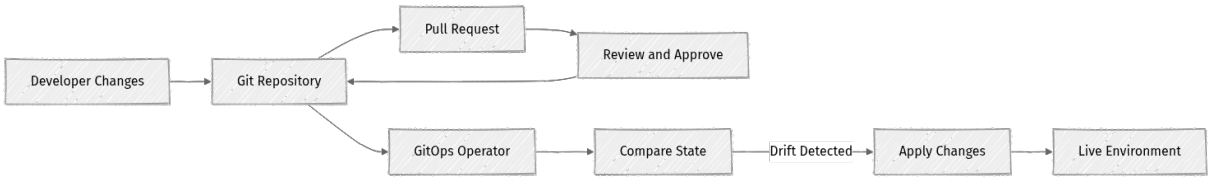
GitOps takes the principles of IaC and CI/CD a step further, making Git the single source of truth for your entire system's desired state.

What is GitOps?

What it is: An operational framework that uses Git to manage infrastructure provisioning and application deployment. You declare the desired state of your infrastructure and applications in Git, and an automated process ensures the live state matches the Git repository.

How it works:

1. **Declarative Configuration:** All infrastructure (IaC) and application deployment configurations (e.g., Kubernetes manifests) are stored in Git.
2. **Pull Requests:** Any change to the desired state is made via a pull request to the Git repository. This allows for code reviews, automated checks, and audit trails.
3. **Automated Reconciliation:** A specialized agent or controller (like Flux or Argo CD for Kubernetes, which are widely used as of 2026) continuously monitors the Git repository. When a change is detected in Git, it automatically applies those changes to the live environment. It "pulls" the desired state from Git and applies it.



Why it exists:

- **Reliability:** Changes are versioned, reversible, and auditable.
- **Security:** Git provides a strong access control layer.
- **Observability:** The Git history provides a clear record of every change.
- **Faster MTTR (Mean Time To Recovery):** If the live environment deviates, the GitOps operator automatically corrects it, or you can roll back to a previous Git commit.

⚡ **Real-world insight:** GitOps is particularly popular in Kubernetes environments because Kubernetes itself is declarative. GitOps tools like Flux and Argo CD leverage this by continuously synchronizing the cluster state with the configuration defined in Git.

Mini-Challenge: Designing an AI Agent Deployment Strategy

Your team has developed a new AI agent service that provides real-time recommendations to users. This agent is critical; even short downtime can lead to a poor user experience and lost revenue. Updates to the agent's underlying model or logic are frequent, sometimes daily.

Challenge: Design a deployment strategy for this AI agent service.

1. Which deployment strategy (Rolling, Blue/Green, Canary) would you recommend and why?
2. What are the key considerations for backward/forward compatibility?
3. How would you integrate this into a CI/CD pipeline, focusing on the deployment stage?
4. Briefly describe how IaC would support this strategy.

Hint: Think about the balance between speed of iteration, minimizing risk, and the unique challenges of AI model updates (e.g., model drift, performance changes).

What to observe/learn: This exercise helps you weigh the tradeoffs of different strategies in a real-world scenario. You'll solidify your understanding of how these concepts intertwine to create a resilient deployment pipeline.

Common Pitfalls & Troubleshooting

Even with the best intentions, automation and deployment strategies can introduce their own set of challenges.

1. **Over-Engineering Automation:** Not everything needs to be automated immediately. Automating a process that changes rarely or is overly complex might be a waste of effort. Focus on repetitive, error-prone tasks first.
    - **Troubleshooting:** Start small. Automate the most painful manual processes. Incrementally add automation as you gain experience and identify further bottlenecks.
2. **Configuration Drift:** Manual changes to infrastructure outside of IaC can cause the live environment to diverge from your code repository. This leads to unexpected behavior and makes subsequent IaC deployments fail.
    - **Troubleshooting:** Enforce IaC strictly. Use tools that can detect drift and report it. Implement automated checks that periodically compare the desired state in Git with the actual state in the cloud.
3. **Inadequate Testing of Deployment Pipelines:** An automated pipeline is only as good as its tests. A broken pipeline can prevent critical updates from reaching production or even cause outages.
    - **Troubleshooting:** Treat your pipeline code (e.g., Jenkinsfiles, GitHub Actions workflows) as critically as your application code. Version control it, review it, and test it in dedicated pipeline testing environments if possible. Ensure rollback mechanisms are also tested.
4. **Ignoring Backward/Forward Compatibility:** Especially with Rolling Updates or Canary deployments, old and new versions of your services might run concurrently. If their APIs or data schemas are incompatible, you'll encounter errors.
    - **Troubleshooting:** Design APIs to be backward compatible. Implement robust versioning strategies. Plan database schema changes carefully to support both old and new application versions during transition periods (e.g., additive changes first, then remove old columns in a subsequent deployment).

Summary

In this chapter, we've explored the critical role of infrastructure automation and modern deployment strategies in building and maintaining scalable, resilient distributed systems:

- **Infrastructure as Code (IaC)** defines your infrastructure declaratively, bringing consistency, speed, and version control to provisioning.
- **Deployment Strategies** like Rolling Updates, Blue/Green, and Canary deployments offer different ways to introduce changes safely, minimizing downtime and risk.
- **Continuous Integration (CI)** ensures code is frequently integrated and tested, while **Continuous Delivery (CD)** automates the process of preparing and optionally deploying changes to production.
- **GitOps** leverages Git as the central source of truth for both infrastructure and applications, enabling reliable, auditable, and automated operations.

These timeless engineering principles are foundational for managing the complexity of modern systems, especially as we integrate sophisticated AI agents that require rapid iteration and robust deployment pipelines. By embracing automation, you move from reactive firefighting to proactive, controlled system evolution.

Next, we'll dive into **Observability**, learning how to understand the internal state of your distributed systems through logging, metrics, and tracing, which is essential for validating your deployments and quickly troubleshooting issues.

References

- Microservices Architecture Style - Azure Architecture Center
- What is Infrastructure as Code? - AWS
- Continuous Integration and Delivery (CI/CD) - Google Cloud
- Blue/Green Deployment - Martin Fowler
- Canary Release - Martin Fowler
- What is GitOps? - FluxCD
- PostgreSQL 15 Release Notes - PostgreSQL Official Documentation

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 11

Advanced Scalability: Caching, Data Consistency, and Distributed Transactions

Welcome back, aspiring system architect! As applications grow and serve more users, the simple solutions of yesterday often hit a wall. In our journey to build robust, scalable systems, we inevitably confront challenges like making data faster to access, keeping it correct across many services, and ensuring complex operations either fully succeed or completely fail.

This chapter dives into three critical, often intertwined, concepts for advanced scalability: **caching strategies**, **data consistency models**, and **distributed transactions**. These are not just theoretical ideas; they are the bedrock of high-performance, reliable systems that handle millions of requests daily. We'll explore timeless principles, understand their practical implications, and learn when to apply them—and critically, when not to.

By the end of this chapter, you'll have a solid conceptual understanding of how to make informed decisions about data management in complex distributed environments, including those involving sophisticated AI agents. This builds directly on our previous discussions about service communication, resilience, and asynchronous workflows, preparing you to design systems that truly stand the test of scale.

The Need for Speed: Mastering Caching Strategies

Imagine your application's most frequently requested data sitting far away, perhaps in a database on another continent. Every request means waiting for network round-trips and database lookups. This latency adds up quickly, making your application feel sluggish.

📌 **Key Idea:** Caching is about storing copies of data closer to where it's needed, reducing latency and database load.

What is Caching and Why Does It Matter?

Caching involves storing frequently accessed data in a faster, more accessible location than its primary source. Think of it like remembering the answers to common questions so you don't have to look them up every time.

Why it exists:

- **Performance:** Reduces latency by serving data from a fast-access memory store (RAM, SSD).
- **Scalability:** Offloads load from primary data stores (databases), allowing them to handle more writes or fewer reads.
- **Cost Reduction:** Less load on databases can mean smaller database instances or fewer read replicas, saving infrastructure costs.

What problem it solves:

- High latency for frequently accessed data.
- Overloading primary data stores with repetitive requests.

A classic example is an API endpoint that fetches popular product listings. Without caching, every user request hits the database. With caching, the first request fetches from the database, stores the result in a cache, and subsequent requests are served almost instantly from the cache.

Common Caching Patterns

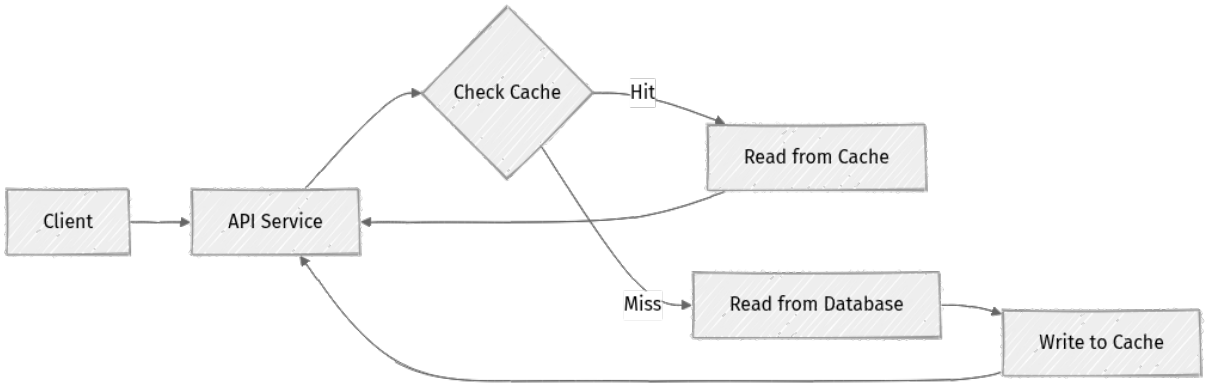
Let's explore the most common ways to integrate a cache into your application. Each pattern has its own strengths and weaknesses, making the choice dependent on your specific workload and consistency requirements.

1. Cache-Aside (Lazy Loading)

This is the most common and often simplest caching pattern. The application code is responsible for checking the cache first.

How it works:

1. **Read request:** The application first checks if the data exists in the cache.
2. **Cache Hit:** If found, the data is returned immediately from the cache. This is the fastest path!
3. **Cache Miss:** If not found (a "miss"), the application fetches the data from the primary data store (e.g., database).
4. **Populate Cache:** After retrieving, the application stores the data in the cache for future requests.
5. **Return Data:** The data is then returned to the client.



Pros:

- **Simplicity:** Easy to implement and understand.
- **No stale data on read:** Only requested data is cached, reducing the chance of caching unneeded or stale data.
- **Resilience:** If the cache fails, the application can still fall back to the database, ensuring availability.

Cons:

- **Initial latency:** The first request for data (a cache miss) will always incur the database lookup latency.
- **Thundering herd problem:** If many requests for the same uncached data arrive simultaneously, they can all hit the database, causing a spike in load.
- **Stale data on updates:** If the underlying data changes in the database, the cache might hold stale data until it expires or is explicitly invalidated.

2. Write-Through

In this pattern, data is written simultaneously to both the cache and the primary data store.

How it works:

1. **Write request:** The application writes data to the cache.
2. **Cache Write:** The cache then immediately writes the same data to the primary data store.
3. **Acknowledge:** Once both writes are complete, the cache (or application) acknowledges the operation.
4. **Read:** Subsequent reads for this data will be served directly from the cache, guaranteed to be fresh.

Pros:

- **Strong consistency on writes:** Data in the cache is always consistent with the primary data store after a write.
- **Simpler read logic:** Reads are always served from the cache (assuming no eviction) and are fresh.

Cons:

- **Higher write latency:** Every write operation incurs the latency of writing to both the cache and the primary data store.
- **Cache can become full:** If not all written data is frequently read, the cache might fill up with unused data, wasting resources.
- **Cache failure:** If the cache fails, writes might be blocked or lost, depending on implementation.

3. Write-Back (Write-Behind)

This pattern prioritizes write performance by writing data to the cache first and asynchronously writing it to the primary data store later.

How it works:

1. **Write request:** The application writes data to the cache.
2. **Acknowledge:** The cache immediately acknowledges the write operation, returning control to the application.
3. **Asynchronous Write:** The cache then asynchronously writes the data to the primary data store in the background. This might happen after a delay, in batches, or when the data is evicted from the cache.

Pros:

- **Very low write latency:** Writes are extremely fast as they only hit the cache.
- **High write throughput:** Can absorb write bursts and batch updates to the primary data store, improving efficiency.

Cons:

- **Data loss risk:** If the cache fails before data is persisted to the primary store, data can be lost. This is a critical consideration.
- **Eventual consistency:** Data in the primary store is eventually consistent with the cache, not immediately.

- **Complexity:** More complex to implement due to asynchronous nature, error handling, and recovery mechanisms.

Cache Invalidation and Eviction

The biggest challenge in caching is often **cache invalidation**—knowing when data in the cache is no longer fresh and needs to be updated or removed. This is where many caching strategies fail if not carefully designed.

Invalidation Strategies:

- **Time-To-Live (TTL):** Data expires after a set duration (e.g., 5 minutes). Simple, but might serve stale data until expiration or fetch fresh data unnecessarily if not expired.
- **Explicit Invalidation:** When data changes in the primary store, the application explicitly tells the cache to remove or update the cached entry. This requires careful coordination between services.
- **Version Numbers:** Store a version number with cached data. When data is updated, increment the version. Reads check if the cached version matches the primary store's version, fetching new if different.

Eviction Policies (when cache is full): When your cache runs out of space, it needs to decide what data to remove.

- **Least Recently Used (LRU):** Removes the item that hasn't been accessed for the longest time, assuming it's less likely to be needed again.
- **Least Frequently Used (LFU):** Removes the item that has been accessed the fewest times, prioritizing frequently used data.
- **First-In, First-Out (FIFO):** Removes the oldest item, regardless of access frequency.

⚡ **Real-world insight:** Many large-scale systems use a combination of these. For instance, a CDN (Content Delivery Network) uses TTL for static assets, while a backend microservice might use explicit invalidation for user profile data that changes frequently.

AI/Agent Workflows and Caching

AI agents often perform computationally expensive tasks or interact with external APIs. Caching is crucial here to improve performance and reduce operational costs:

- **Model Inference Results:** Cache the output of an expensive AI model inference for a given input. If the same input comes again, serving from cache avoids re-running the model, saving compute cycles.
- **API Responses:** If an agent queries a third-party API, cache its responses to avoid rate limits and reduce latency, especially for common queries.
- **Intermediate Computations:** Agents might generate intermediate data structures or processing results that can be reused across different steps or agents in a complex workflow.
- **Knowledge Base Entries:** Cache frequently accessed entries from a vector database or knowledge store to speed up RAG (Retrieval Augmented Generation) processes, significantly reducing query times.

The Consistency Conundrum: Data Consistency in Distributed Systems

In a single-server application with one database, data consistency is relatively straightforward. When you update a record, everyone sees the latest version. In distributed systems, where data might be replicated across multiple servers or sharded across different databases, ensuring consistency becomes a fundamental challenge.

🧠 **Important:** You can't always have everything you want in distributed systems. You must make tradeoffs between consistency, availability, and partition tolerance.

The CAP Theorem (Briefly)

The CAP theorem states that a distributed data store can only simultaneously guarantee two out of three properties:

1. **Consistency (C):** All clients see the same data at the same time, regardless of which node they connect to.
2. **Availability (A):** Every request receives a response, without guarantee that it contains the most recent version of the information.
3. **Partition Tolerance (P):** The system continues to operate despite arbitrary message loss or failure of parts of the system (network partitions).

In a distributed system, network partitions are inevitable. Therefore, you must always design for **Partition Tolerance (P)**. This means you are forced to choose between **Consistency (C)** and **Availability (A)** during a network partition.

- **CP System:** Prioritizes consistency over availability. If a partition occurs, the system will block or return an error until consistency can be guaranteed. Examples include traditional relational databases with strong consistency guarantees, or systems using consensus algorithms like Paxos/Raft.
- **AP System:** Prioritizes availability over consistency. If a partition occurs, the system will remain available but might return stale data. Consistency is eventually achieved once the partition heals. Examples include many NoSQL databases like Cassandra or DynamoDB.

Understanding CAP helps you choose the right data store and consistency model for different parts of your system. There's no one-size-fits-all answer.

Consistency Models

Different applications have different consistency requirements. Deciding which model to use is a critical architectural decision.

1. Strong Consistency

In a strongly consistent system, once a write operation is complete, any subsequent read operation is guaranteed to see that updated value. It's like everyone reading the same book at the exact same page, always seeing the latest edits.

When it's needed:

- Financial transactions (e.g., bank account balances, preventing overdrafts).
- Inventory management (e.g., ensuring an item is not sold twice).
- User authentication and authorization (e.g., ensuring a password change is immediately active).

How it's achieved (conceptually):

- **Distributed Locks:** Ensuring only one writer can modify a piece of data at a time across multiple nodes.
- **Consensus Algorithms:** Such as Paxos or Raft, which ensure all nodes agree on the order of operations and the state of the data. These are complex and add latency.

Tradeoffs: Higher latency for writes, potentially lower availability during network partitions, and more complex to implement and manage.

2. Eventual Consistency

In an eventually consistent system, after a write operation, the data might not be immediately visible to all readers. There's a delay, but eventually, all replicas will converge to the same state. It's like everyone eventually getting the latest edition of a newspaper; there might be a brief period where some have an older version.

When it's acceptable:

- Social media likes or comment counts (a slight delay in seeing the latest count is fine).
- Shopping cart contents (minor inconsistencies are tolerable, users can refresh).
- User profile updates (it's okay if a new profile picture takes a few seconds to propagate globally).
- AI agent internal state that can be reconciled later without immediate critical impact.

How it's achieved (conceptually):

- **Asynchronous Replication:** Changes are propagated between nodes in the background, often via message queues or replication streams.
- **Conflict Resolution:** If conflicts arise (two nodes update the same data differently), rules are in place to resolve them (e.g., "last writer wins," application-specific logic, or Conflict-free Replicated Data Types (CRDTs)).

Tradeoffs: Lower latency for writes, higher availability during partitions, and often simpler to scale. The main challenge is managing and understanding the eventual nature of consistency, and designing your application to tolerate brief periods of inconsistency.

⚡ **Real-world insight:** Most large-scale distributed systems, especially those prioritizing availability, leverage eventual consistency for many of their components. For example, a global AI service might store user preferences with eventual consistency for faster access, while billing information requires strong consistency.

AI/Agent Workflows and Consistency

Consistency is paramount when multiple AI agents collaborate or when an agent's actions have real-world implications.

- **Collaborative Agents:** If multiple agents are working on a shared knowledge base or task, ensuring consistent views of that shared state is crucial. Eventual consistency might be acceptable for transient states, but strong consistency might be needed for critical decisions or final output.
- **Agent Actions:** If an AI agent initiates a real-world action (e.g., placing an order, sending an email), the system needs to know if that action truly occurred. This might involve strong consistency checks or transactional guarantees.
- **State Synchronization:** Agents often maintain internal state. When this state needs to be synchronized or shared across different instances or agents, the appropriate consistency model must be chosen based on the criticality of that state.

The Atomic Challenge: Distributed Transactions

Sometimes, an operation isn't just one step; it's a sequence of interdependent steps that must either all succeed or all fail together. This is the essence of a transaction. In a distributed system, where these steps might involve multiple services and databases, we're talking about **distributed transactions**.

The Problem with Distributed Transactions

Traditional ACID (Atomicity, Consistency, Isolation, Durability) transactions are designed for a single database. In a distributed system, achieving ACID properties across multiple, independent services is incredibly difficult and often comes with severe performance and availability penalties.

Why it's hard:

- **Network Latency:** Coordinating across multiple services involves many network calls, significantly increasing overall transaction time.
- **Partial Failures:** What if one service commits, but another fails? How do you roll back the first service, which might have already committed its part?
- **Concurrency:** Managing locks across multiple services can lead to deadlocks and reduced throughput, severely limiting scalability.

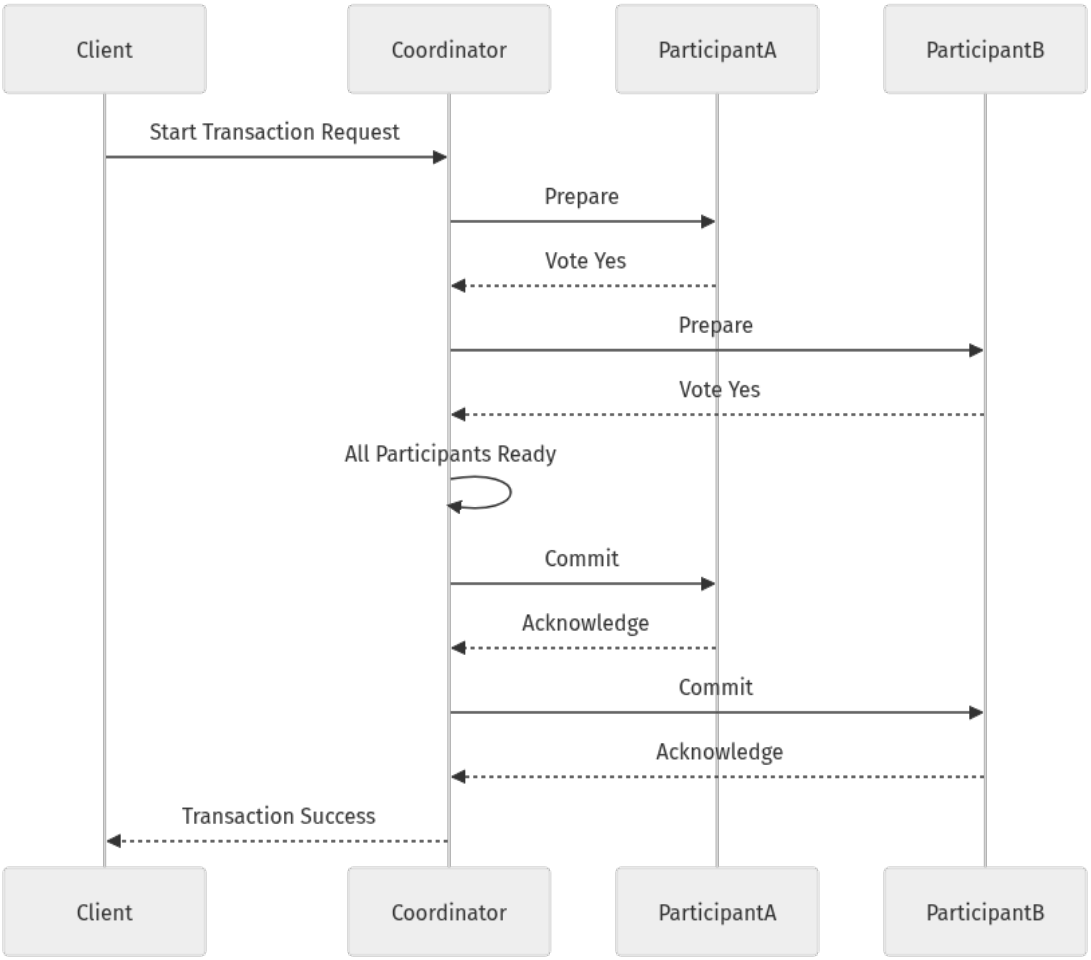
Because of these complexities, the general advice in modern distributed system design is to **avoid distributed transactions if possible.** Instead, favor single-service transactions and eventual consistency. However, there are scenarios where some form of transactional guarantee is necessary.

1. Two-Phase Commit (2PC)

2PC is a classic protocol designed to provide atomic transactions across distributed resources. It's a heavyweight solution for strong consistency.

How it works: It involves a **coordinator** (often a transaction manager) and several **participants** (services or databases).

1. **Phase 1: Prepare (Vote)**
    - The coordinator sends a "prepare" message to all participants, asking them to prepare to commit the transaction.
    - Each participant performs its local transaction, writes necessary logs to disk (making it durable), and responds "yes" (ready to commit) or "no" (cannot commit).
2. **Phase 2: Commit / Rollback**
    - **If all participants respond "yes":** The coordinator sends a "commit" message to all participants. Each participant then commits its local transaction.
    - **If any participant responds "no" (or times out):** The coordinator sends a "rollback" message to all participants. Each participant then undoes its local transaction.



Pros:

- Provides strong transactional guarantees (ACID) across distributed resources.

Cons:

- **Blocking:** Participants hold locks and resources during both phases, leading to poor concurrency and high latency.
- **Single Point of Failure:** The coordinator is a critical component. If it fails during the commit phase, participants might be left in an uncertain state (requiring complex recovery heuristics).
- **High Latency:** Multiple network round-trips and disk I/O make it inherently slow.
- **Scalability Bottleneck:** Limits throughput due to its blocking and centralized nature.

Verdict: Rarely used in modern, highly scalable distributed systems due to its severe performance and availability drawbacks. It's often a last resort for very specific strong consistency needs in tightly coupled enterprise systems.

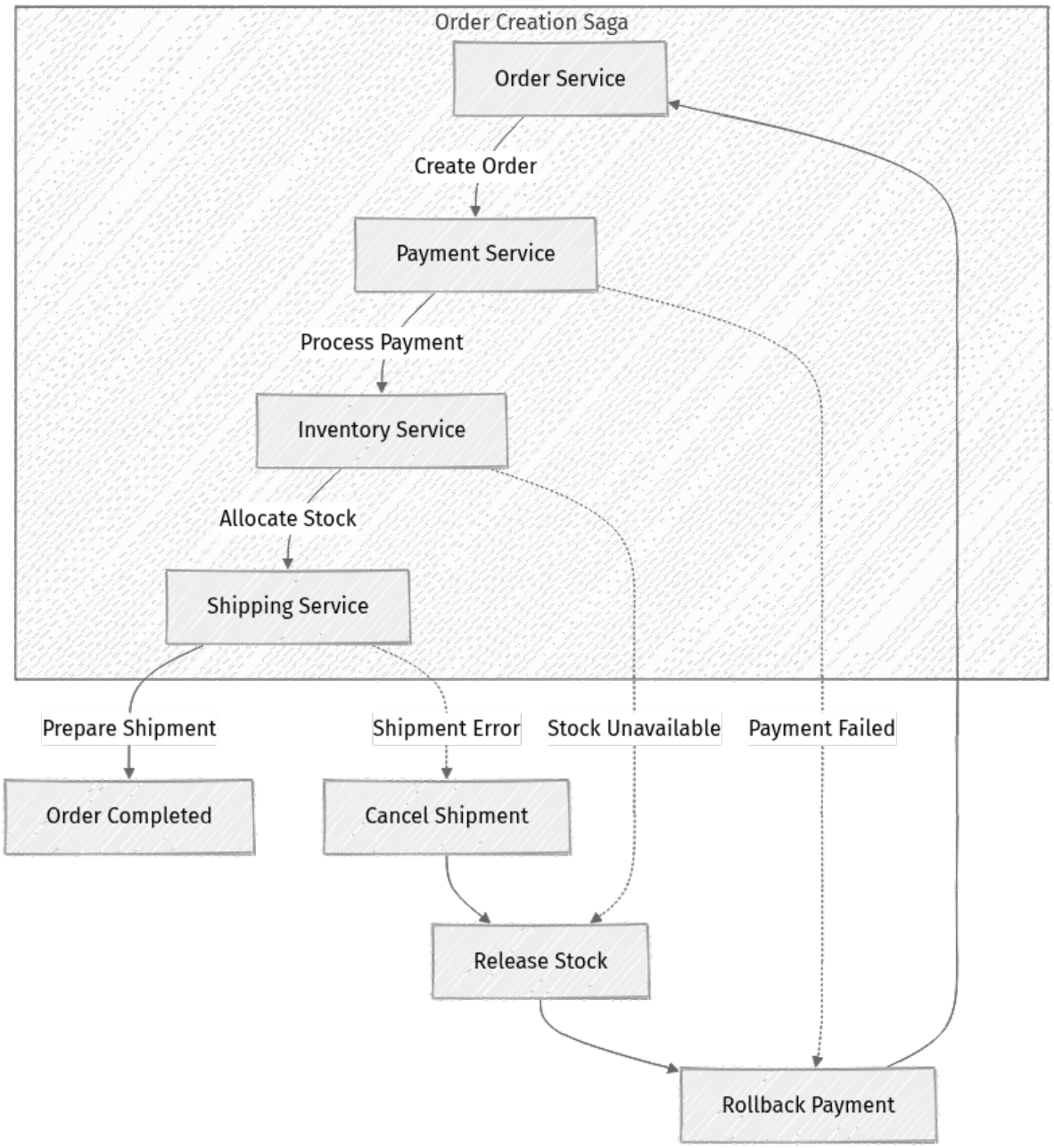
2. Saga Pattern

The Saga pattern is an alternative to 2PC for managing long-running, distributed transactions. Instead of a single atomic transaction, a Saga is a sequence of local transactions, where each local transaction updates data within a single service. If a step fails, compensation transactions are executed to undo the effects of previous steps. The system achieves eventual consistency for the overall operation.

How it works: A Saga can be implemented in two main ways:

1. **Choreography:** Services publish events, and other services subscribe to these events to perform their next local transaction. This is a decentralized approach, where services react to events without a central coordinator.
2. **Orchestration:** A central orchestrator (a dedicated service) manages the sequence of local transactions and triggers compensation actions if necessary. The orchestrator is responsible for the overall flow.

Let's illustrate with an Orchestration-based Saga for an e-commerce order:



Pros:

- **High Availability:** No single point of failure like a 2PC coordinator.
- **Better Scalability:** Services can commit local transactions quickly, releasing locks sooner.
- **Loosely Coupled:** Services interact via events or commands, making them more independent and resilient.

Cons:

- **Eventual Consistency:** The overall saga is eventually consistent, not strongly consistent at every step.

- **Complexity:** Managing compensation logic can be intricate, especially for complex workflows and error handling.
- **Debugging:** Tracing a saga across multiple services can be challenging due as it spans multiple services and potentially long durations.

Verdict: The Saga pattern is the preferred approach for distributed transactions in modern microservice architectures, accepting eventual consistency in favor of higher availability and scalability.

AI/Agent Workflows and Distributed Transactions

For AI agents, transactional guarantees are critical when actions have irreversible or high-stakes consequences.

- **Atomic AI Agent Pipelines:** If an AI agent workflow involves multiple steps (e.g., data retrieval, processing, model inference, external API call) and they must all succeed or fail together, a Saga-like pattern is ideal. For example, an agent that books travel might need to reserve a flight, book a hotel, and confirm payment. If any step fails, previous steps must be compensated to avoid an inconsistent state (e.g., a booked flight but no hotel).
- **Resource Allocation:** When an AI agent allocates a limited resource (e.g., GPU time, specific compute instances) across multiple requests, ensuring that allocation is atomic and consistent is vital. A Saga could manage the reservation and release of these resources.
- **Financial Transactions:** Any AI agent dealing with monetary transactions absolutely requires robust transactional guarantees, likely through a Saga pattern that integrates with financial microservices, where each local transaction is handled by a dedicated financial service.

Applying the Principles: A Guided Design Exercise

Let's put these concepts into practice by designing a simplified AI-driven customer support system. Imagine a system where users submit support tickets, an AI agent processes them, fetches relevant information, and potentially takes actions.

Scenario: An AI Customer Support Agent processes incoming tickets.

1. User submits a ticket via a web portal.
2. `Ticket Service` receives the ticket and stores it.

3. `AI Orchestrator` picks up the ticket.
4. `Knowledge Base Agent` queries a vector database for relevant FAQs and articles.
5. `Sentiment Analysis Agent` analyzes the ticket's tone.
6. `Action Agent` (if appropriate) attempts to resolve the issue by interacting with an external system (e.g., resetting a password via an `Auth Service`).
7. Finally, the `Ticket Service` updates the ticket status and adds agent notes.

Consider how you would incorporate caching, consistency, and transactional guarantees into this workflow.

Step 1: Optimizing Knowledge Base Access with Caching

The `Knowledge Base Agent` frequently queries the vector database for common issues. This can be slow and expensive.

Your Design Thought Process:

- **Problem:** High latency and load on the vector database for common queries.
- **Solution:** Introduce a cache.
- **Which Caching Pattern?** `Cache-Aside` makes the most sense here. Why? Because the `Knowledge Base Agent` needs to check for freshness, and if not found, retrieve from the database. This avoids caching irrelevant data.
- **Invalidation Strategy?** `TTL` (Time-To-Live) for general knowledge base articles is a good start. Perhaps 1-2 hours. If a critical article is updated, an `Explicit Invalidation` message could be sent to the cache.

Guided Exercise: How would you design the `Knowledge Base Agent` to use a `Cache-Aside` pattern with a `TTL` of 1 hour for standard queries?

- **Consider:** What happens on a cache hit? What on a miss? When does data expire?

Step 2: Ensuring Consistency for Ticket Status

The `Ticket Service` needs to maintain the current status of a ticket (e.g., "New," "Processing," "Resolved"). Multiple agents might try to update the status or read it.

Your Design Thought Process:

- **Problem:** Ensuring all parts of the system see the correct, latest ticket status.
- **Consistency Requirement:** Does the ticket status need to be strongly consistent (everyone sees the update instantly) or is eventually consistent acceptable (a brief delay is okay)? For a customer support ticket, strong consistency is generally preferred to avoid agents working on outdated information.
- **Solution:** Use a strongly consistent data store for the primary ticket status.
- **Tradeoffs:** This might mean slightly higher latency for status updates, but it prevents major operational errors.

Guided Exercise: If the `Ticket Service` uses a relational database, how does it inherently support strong consistency for a single ticket's status? What mechanisms are in place (e.g., database transactions, isolation levels)?

Step 3: Handling the "Action Agent" Workflow with Transactional Guarantees

The `Action Agent` attempting to reset a password via the `Auth Service` is a critical operation. If the password reset succeeds, the ticket status must be marked "Resolved" and notes added. If the reset fails, the ticket status must not be marked "Resolved", and failure notes must be added. This is a distributed transaction.

Your Design Thought Process:

- **Problem:** Multiple services (Action Agent, Auth Service, Ticket Service) involved in an atomic operation.
- **Transactional Requirement:** All steps must succeed or all fail. `2PC` is too heavy. The `Saga Pattern` is the modern approach.
- **Solution:** Implement an Orchestration-based Saga.
- **Saga Steps:**
    1. `AI Orchestrator` initiates "Password Reset Saga."
    2. `Action Agent` calls `Auth Service` to reset password.
    3. If `Auth Service` succeeds, `Action Agent` notifies `Ticket Service` to update status to "Resolved" and add success notes.
    4. If `Auth Service` fails, `Action Agent` notifies `Ticket Service` to update status to "Failed" and add error notes (this is the compensation action for the overall saga failure).

- **Compensation:** If the `Auth Service` fails, no compensation is needed for the `Auth Service` itself (it didn't commit anything). The `Ticket Service` simply logs the failure.

Guided Exercise: Draw a simple Mermaid `sequenceDiagram` or `flowchart TD` (max 8 nodes) illustrating this Password Reset Saga. Focus on the happy path and one failure path, showing the services involved and the messages exchanged.

Mini-Challenge: Caching for an AI Agent Marketplace

Imagine you're building an AI Agent Marketplace where users can browse, purchase, and deploy various AI agents.

Challenge:

1. Identify at least two pieces of data in this marketplace that would benefit significantly from caching.
2. For each piece of data, propose a caching strategy (Cache-Aside, Write-Through, or Write-Back) and an invalidation strategy. Justify your choices based on performance, consistency needs, and potential for staleness.

Hint: Think about data that changes infrequently but is read often, versus data that needs to be highly consistent. Consider both read-heavy and write-heavy scenarios.

Common Pitfalls & Troubleshooting

Building scalable and consistent distributed systems is challenging. Here are some common traps:

1. **Cache Invalidation Nightmares:** The classic "two hard problems in computer science are cache invalidation, naming things, and off-by-one errors." If your invalidation strategy is flawed, users will see stale data, leading to confusion or incorrect behavior.
    - **Troubleshooting:** Implement robust monitoring for cache hits/misses, data freshness, and cache size. For critical data, use explicit invalidation triggered by database updates. For less critical data, a reasonable TTL is usually fine.

2. **Over-Engineering Consistency:** Blindly aiming for strong consistency everywhere can cripple your system's performance and scalability. Each strong consistency guarantee adds overhead.
    - **Troubleshooting:** Carefully analyze the consistency requirements for each piece of data. Ask: "What happens if this data is briefly stale? Is it acceptable? What's the business impact?" Most of the time, eventual consistency is sufficient and vastly more scalable.
3. **Premature Distributed Transactions (2PC):** Jumping to a complex 2PC implementation when simpler patterns (like queues with retries and idempotency, or the Saga pattern with eventual consistency) would suffice.
    - **Troubleshooting:** Always explore simpler options first. Can you design your services so that operations are idempotent, allowing for safe retries? Can you tolerate eventual consistency for the overall flow? Only consider heavyweight options if business requirements absolutely demand strong, global atomicity.
4. **Ignoring Cache Warming:** A newly deployed service or an empty cache can lead to a "cold start" problem where the first requests hit the database, causing a performance spike.
    - **Troubleshooting:** Consider cache warming strategies for critical data, where you pre-populate the cache with frequently accessed items, especially after deployments or cache resets.

Summary

In this chapter, we've explored advanced techniques crucial for scaling and ensuring correctness in distributed systems:

- **Caching** is essential for reducing latency and database load. We discussed **Cache-Aside**, **Write-Through**, and **Write-Back** patterns, along with strategies for **invalidation** and **eviction**.
- **Data Consistency** in distributed systems forces tradeoffs, as highlighted by the **CAP Theorem**. We differentiated between **Strong Consistency** (high guarantees, high latency) and **Eventual Consistency** (lower guarantees, higher availability), understanding when to apply each.

- **Distributed Transactions** are complex due to the inherent nature of distributed systems. We learned why **Two-Phase Commit (2PC)** is generally avoided in favor of patterns like the **Saga Pattern**, which uses a sequence of local transactions and compensation actions to achieve eventual consistency with better scalability.
- We also saw how these principles apply to the unique challenges and opportunities in building resilient and intelligent **AI/Agent workflows**, particularly in scenarios requiring fast access to information or coordinated actions.

Understanding these concepts allows you to make informed architectural decisions, balancing performance, availability, and data integrity.

What's Next?

With our systems becoming increasingly complex and distributed, another crucial aspect emerges: ensuring they are secure and cost-effective. In our next chapter, we'll delve into **Security and Cost Optimization in Distributed Systems**, exploring how to protect your applications from threats and manage resource consumption efficiently.

References

- Microsoft Azure Architecture Center - Caching guidance
- Microsoft Azure Architecture Center - Data consistency primer
- Microsoft Azure Architecture Center - Saga pattern
- Martin Fowler - Saga
- Wikipedia - CAP theorem

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 12

Systems Thinking, Tradeoffs, and Architecting for AI/Agentic Workflows

In the journey from a simple application to a complex distributed system, we've explored many patterns and practices. Yet, the most powerful tool in an engineer's arsenal isn't a specific technology or framework—it's a way of thinking. This chapter brings it all together, focusing on systems thinking, the art of navigating architectural tradeoffs, and how these timeless principles are more critical than ever when building the next generation of AI and agentic workflows.

You've already built a solid foundation in understanding individual components like reverse proxies, service communication, and asynchronous patterns. Now, we'll elevate that understanding to view your entire architecture as a living, breathing system. This shift in perspective is essential for designing resilient, scalable, and maintainable systems, especially as we integrate intelligent agents that interact autonomously.

The Holistic View: Mastering Systems Thinking

At its heart, systems thinking is about understanding how individual components interact to form a larger, complex whole. It’s moving beyond optimizing a single service to considering its ripple effects across the entire ecosystem.

Why it matters: Without a holistic view, you might make a local optimization that inadvertently creates a bottleneck or introduces instability elsewhere. Imagine tuning a single engine component for maximum power without considering how it affects the car's transmission, cooling, or fuel efficiency. The goal isn't just to make one part fast, but to make the entire system perform optimally and reliably.

📌 **Key Idea:** Systems thinking helps us see the forest and the trees, ensuring local decisions contribute positively to global system goals.

This approach becomes vital when designing distributed systems, where partial failures are inevitable, and interactions are complex. It helps you anticipate cascading failures, design for resilience, and understand the true cost of architectural decisions.

Navigating Architectural Tradeoffs: The Art of Compromise

There's no such thing as a perfect system. Every architectural decision involves a tradeoff. Understanding these tradeoffs is the bedrock of good systems design. Your role as an architect isn't to eliminate tradeoffs, but to manage them effectively, aligning decisions with business goals and constraints.

Let's explore some common tradeoffs you'll encounter and how to think about them:

- **Consistency vs. Availability (CAP Theorem):** In a distributed system, you can only choose two out of three: Consistency, Availability, or Partition Tolerance. Since network partitions are a given in distributed systems, you're always choosing between strong consistency (all nodes see the same data at the same time) and high availability (the system remains operational even if some nodes fail).
    - **Real-world insight:** A banking system prioritizes strong consistency for transactions. A social media feed might prioritize availability and eventual consistency. You must decide which is more critical for your specific use case.
- **Performance vs. Cost:** Achieving ultra-low latency or extremely high throughput often requires more expensive infrastructure (e.g., faster CPUs, more RAM, premium network bandwidth, specialized hardware).
    - **Real-world insight:** A real-time bidding system will invest heavily in performance, accepting higher costs. An overnight batch processing job, however, might optimize for cost, tolerating longer execution times.
- **Complexity vs. Flexibility:** More abstract, modular, or generalized designs can be highly flexible and adaptable but often introduce significant complexity in initial development and ongoing maintenance. Simpler designs might be faster to build but harder to adapt to future changes.
    - **Real-world insight:** A microservices architecture offers high flexibility and independent scaling but is inherently more complex to develop, deploy, and operate than a monolithic application.

- **Time-to-Market vs. Long-Term Maintainability:** Rushing a feature out the door can lead to technical debt, making the system harder and more expensive to maintain or evolve later. This is often a strategic business decision.
    - **Real-world insight:** A startup might prioritize rapid iteration to find product-market fit, accepting some technical debt. An established enterprise, especially for critical systems, would prioritize maintainability and stability.
- **Operational Overhead vs. Developer Experience:** Automating infrastructure and operations (DevOps) reduces manual effort and improves reliability but requires upfront investment in tooling and expertise. Sometimes, a simpler deployment model might be easier for developers initially, but harder to operate at scale.
    - **Real-world insight:** Implementing a robust CI/CD pipeline might be a significant upfront effort but pays dividends in reduced operational burden and faster, more reliable deployments over time.

🧠 **Important:** Good architectural decisions are context-dependent. What's right for one system or business might be wrong for another. There's no universal "best" approach. Your decisions should always align with your business goals and constraints.

Architecting for AI/Agentic Workflows: Timeless Principles Applied

AI and agentic systems, by their very nature, are often distributed and complex. An "agent" is essentially a specialized service that can perceive, reason, and act autonomously or semi-autonomously. These systems leverage all the distributed system patterns we've discussed, but with unique considerations for managing intelligence, compute, and interaction.

1. Decomposition for Agents: Microservices for Intelligence

Just as a large application is broken into microservices, complex AI tasks can be decomposed into smaller, specialized agents or services. This modularity allows for clearer responsibilities, independent scaling, and fault isolation.

- **Perception Agent:** Handles input (e.g., listening to a conversation, reading a document, parsing sensor data).
- **Reasoning Agent:** Processes information, applies logic, makes decisions, potentially using large language models (LLMs) or expert systems.

- **Action Agent:** Executes tasks (e.g., sending an email, updating a database, calling an external API, controlling a robot).
- **Memory Agent:** Stores and retrieves context, long-term knowledge, and past interactions.
- **Orchestration Agent:** Manages the flow and interaction between other agents, defining the overall workflow.

Each of these can be a distinct service, potentially running different models or computational requirements. This allows for independent scaling, development, and failure isolation, echoing the benefits of microservices.

2. Asynchronous Communication: Agents Talking to Agents

Agents rarely operate in perfect synchronous harmony. They often need to exchange information, trigger subsequent actions, or update shared state without waiting for an immediate response. This is where queues and event-driven systems shine, enabling loose coupling and resilience.

- **Message Queues:** An agent finishes a task (e.g., a `SearchAgent` finds relevant documents) and publishes a message to a queue. An `AnalysisAgent` subscribes to this queue, picks up the message, and starts processing. This pattern ensures that if one agent is temporarily unavailable, messages can queue up and be processed once it recovers, preventing upstream agents from blocking.
- **Event Buses:** For more complex, many-to-many interactions, an event bus allows agents to broadcast events (e.g., `UserQueryReceived`, `TaskCompleted`, `DecisionMade`). Other interested agents can react to these events, enabling dynamic and flexible workflows.

3. Worker Architectures for Compute-Intensive AI

AI models, especially large language models (LLMs) or complex simulation agents, require significant computational resources (GPUs, specialized accelerators). Worker architectures are perfectly suited for handling these variable and often bursty workloads.

- **Dedicated Inference Workers:** Services dedicated to running AI model inference. When a `ReasoningAgent` needs to make a prediction, it sends a request to an inference worker pool via a queue. These workers are often optimized for specific hardware.
- **Batch Processing Workers:** For training models or processing large datasets, long-running worker services can pull tasks from queues, execute compute-heavy jobs, and report results.

- **Dynamic Scaling:** These worker pools can be scaled up or down based on demand, using infrastructure automation to manage costs and performance, crucial for managing fluctuating AI workloads.

4. Observability for Multi-Agent Systems

Debugging a single service is hard; debugging a system where multiple intelligent agents are interacting, making decisions, and potentially failing, is even harder. Robust observability is non-negotiable for understanding, debugging, and improving AI systems.

- **Logging:** Every agent should log its perceptions, reasoning steps, decisions, and actions. This creates an audit trail, vital for understanding why an agent behaved a certain way.
- **Metrics:** Track agent-specific metrics like task completion rates, decision accuracy, latency of model calls, queue depths, and resource utilization. These metrics provide insights into system health and performance.
- **Distributed Tracing:** Crucial for understanding the end-to-end flow of a request or a "thought process" across multiple agents. If a user query triggers a `PerceptionAgent`, which then calls a `ReasoningAgent`, then an `ActionAgent`, tracing links all these operations together, allowing you to pinpoint where delays or failures occur.

Without these, your AI system becomes a "black box," impossible to understand or improve effectively.

5. Resilience in Agentic Systems

Agents can fail, just like any other service. Designing for resilience means anticipating these failures and building mechanisms to recover gracefully.

- **Retries and Backoff:** If an `ActionAgent` fails to call an external API, it should retry with an exponential backoff strategy, preventing overwhelming the failing service.
- **Circuit Breakers:** Prevent an agent from continuously hammering a failing dependency, giving the dependency time to recover and preventing cascading failures.
- **Idempotency:** Agent actions should be idempotent where possible, meaning performing the action multiple times has the same effect as performing it once (e.g., updating a status). This simplifies retry logic.
- **Dead Letter Queues (DLQs):** Messages that agents cannot process after multiple retries should go to a DLQ for manual inspection, preventing them from blocking the main workflow.

6. Infrastructure Automation for AI (MLOps)

Deploying and managing AI models and agent infrastructure is complex and dynamic. Automation is key to ensuring efficiency, consistency, and reliability. This practice is often referred to as MLOps (Machine Learning Operations).

- **Model Deployment Pipelines:** Automate the process of packaging, versioning, and deploying new or updated AI models to inference services. This ensures that models are consistently and safely rolled out.
- **Resource Provisioning:** Automatically spin up and tear down GPU instances or specialized compute clusters based on agent workload demands, optimizing for cost and performance.
- **Experiment Tracking:** Tools that integrate with your infrastructure to track model performance, resource usage, and hyperparameters across different experiments, facilitating continuous improvement.

By embracing these principles, you can build AI and agentic systems that are not just intelligent, but also robust, scalable, and manageable.

Guided Scenario: Building an AI Research Agent Workflow

Let's walk through a conceptual example of a sophisticated AI "Research Agent" designed to answer complex user queries by synthesizing information from multiple sources. We'll see how distributed system patterns are fundamental to its operation.

Step 1: The User Request and Initial Routing

Our scenario begins with a user submitting a query, like "Summarize the latest trends in quantum computing and their potential impact on AI."

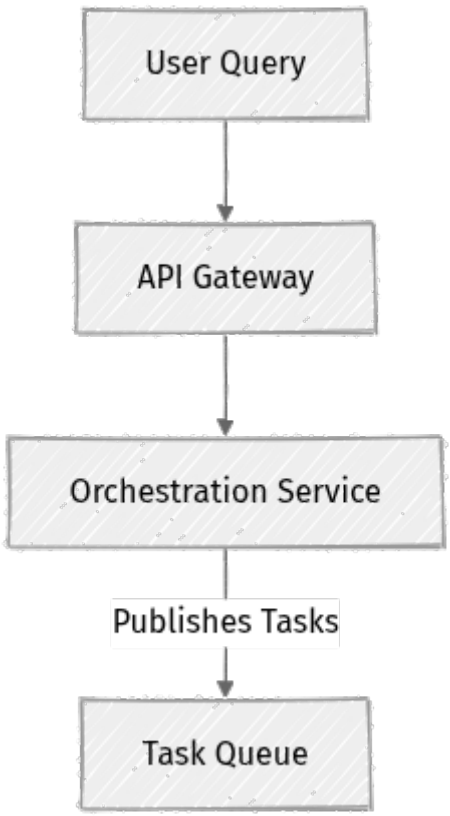
1. **User Interface:** The user types their query into a web application or chat interface.
2. **API Gateway/Reverse Proxy:** The request first hits an API Gateway. This gateway acts as the single entry point, handling authentication, rate limiting, and routing. It then forwards the query to our specialized `Orchestration Service`.
    - **Why a Gateway?** It centralizes common concerns and protects our backend services from direct exposure.

Step 2: Orchestrating the Research Task

The `Orchestration Service` is the brain of our Research Agent, responsible for breaking down the complex user request and coordinating the work of other agents.

1. **Orchestration Service:** This service receives the user's query. It uses its own internal logic or a small language model to break down the query into smaller, manageable sub-tasks. For example, the query "Summarize quantum computing trends and AI impact" might become:
    - "Search for recent quantum computing advancements."
    - "Search for potential AI impacts of quantum computing."
    - "Analyze search results for key themes."
    - "Synthesize findings into a comprehensive report."
2. **Task Queue:** The `Orchestration Service` publishes each of these sub-tasks as messages to a **Task Queue**.
    - **Why a Queue?** This is a critical asynchronous pattern. The orchestrator doesn't wait for each task to complete. It simply places the tasks in a queue and immediately becomes free to handle other user requests. This provides resilience: if a downstream agent is temporarily busy or fails, the messages persist in the queue until they can be processed, preventing bottlenecks and cascading failures.

Here's how this initial flow might look:



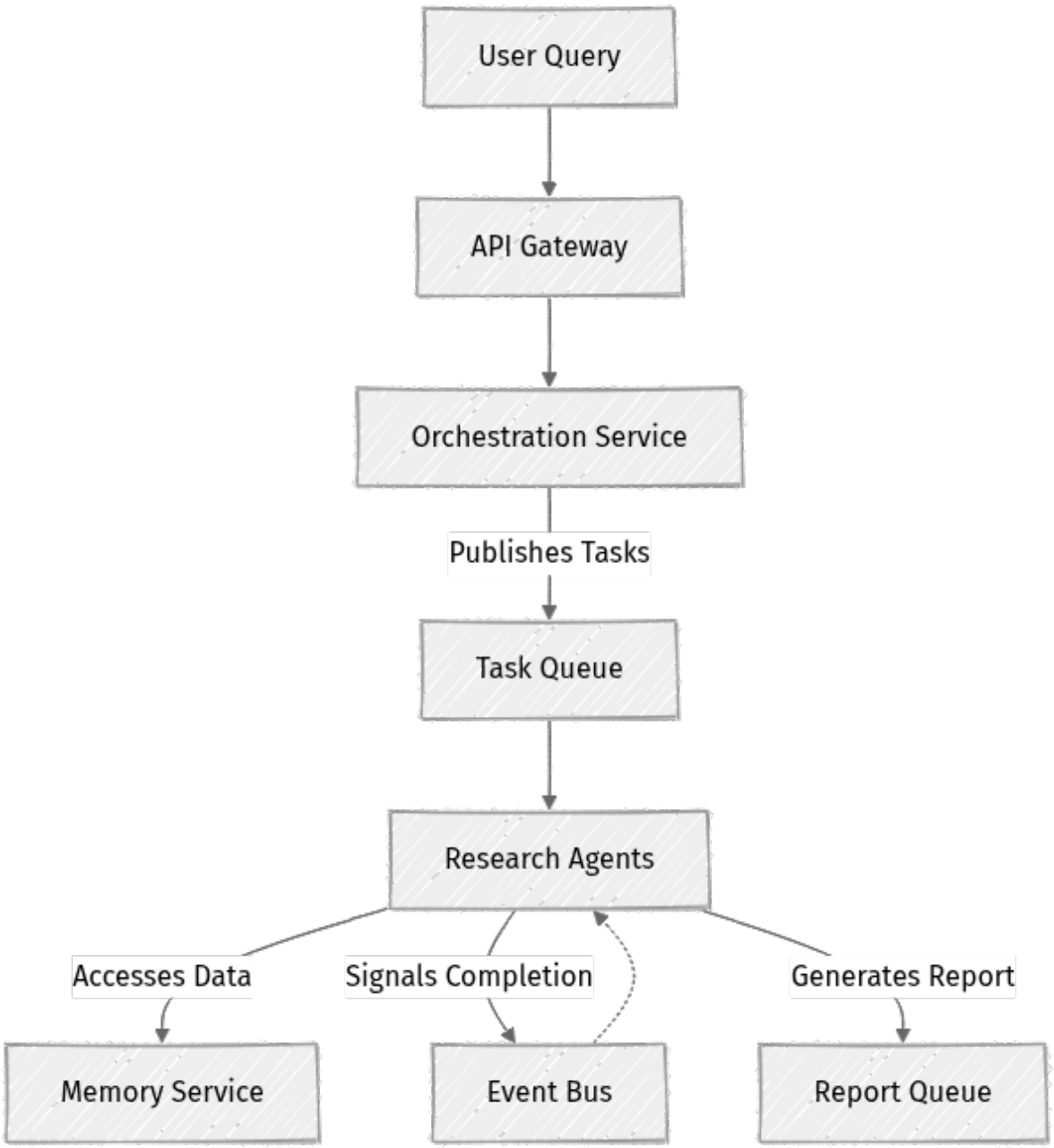
Step 3: Specialized Agents at Work (The Distributed Processing)

Now, various specialized worker agents pick up tasks from the `Task Queue` and perform their specific functions.

1. **Search Agent (Worker Service):**
    - This agent constantly monitors the `Task Queue` for "search" tasks.
    - When it receives a task (e.g., "Search for recent quantum computing advancements"), it uses external search APIs (like Google Scholar, academic databases, news aggregators) to gather information.
    - It's a worker service, meaning we can scale multiple instances of it based on the volume of search tasks.
    - After finding relevant results, it sends them to a **Memory Service** (a high-performance database or key-value store optimized for knowledge retention) for persistent storage.
    - It then publishes an "SearchCompleted" event to an **Event Bus**, signaling that its task is done and the results are available.

2. **Analysis Agent (Worker Service):**
    - This agent subscribes to the `Task Queue` for "analysis" tasks and also listens to "SearchCompleted" events on the **Event Bus**.
    - When triggered, it retrieves the raw search data from the **Memory Service**.
    - It then uses a large language model (LLM) to summarize, extract key points, identify relationships, and perhaps even flag conflicting information. This is a compute-intensive operation.
    - This LLM inference is handled by a pool of dedicated, potentially GPU-accelerated, inference workers.
    - Analyzed data (e.g., key themes, extracted facts) is stored back into the **Memory Service**, and an "AnalysisCompleted" event is published to the **Event Bus**.
3. **Synthesis Agent (Worker Service):**
    - This agent subscribes to the `Task Queue` for "synthesis" tasks and listens for "AnalysisCompleted" events.
    - Once all relevant analysis tasks are complete, it retrieves all processed data from the **Memory Service**.
    - It then uses another LLM to generate a coherent, comprehensive report, synthesizing the findings into a user-friendly format.
    - The final report is stored in the **Memory Service**, and a "ReportReady" event is published to a dedicated **Report Queue**.

Here's an updated diagram showing the interaction of these agents:



Step 4: Delivering the Result

Finally, the completed report needs to be delivered back to the user.

1. **Notification Service:** This service subscribes to the `Report Queue`.
2. When a "ReportReady" event arrives, it fetches the final report from the **Memory Service**.
3. It then delivers the report back to the user, either through the original API Gateway, a real-time notification channel (like WebSockets), or an email.

Observability in Action: Throughout this entire process, every agent logs its actions, success/failure statuses, and latency. Distributed tracing links the initial user query all the way through to the final report delivery, making it possible to diagnose exactly where a delay or error occurred. This allows us to see the "thought process" of our AI Research Agent.

This example illustrates how AI/agentic systems are simply advanced applications of the distributed system patterns we've explored, emphasizing asynchronous communication, specialized services, and robust observability.

Mini-Challenge: Design a Personal Shopping Assistant Agent

Now it's your turn to apply systems thinking to an AI problem.

Challenge: Outline the core components (agents/services) and their communication flow for a "Personal Shopping Assistant Agent" that helps a user find the best deal for a specific product online. Consider what external services it might interact with (e.g., e-commerce sites, price comparison APIs) and how you'd ensure its resilience and scalability.

Think about:

- What's the initial input from the user?
- What different kinds of information does it need to gather (product details, prices, reviews)?
- How would it compare options from various sources?
- How would it present the final recommendation to the user?
- Where might queues or event buses be useful to manage asynchronous operations or handle failures?
- What role would a "Memory Service" play in remembering user preferences or past searches?

What to observe/learn: This exercise helps you practice decomposing a complex problem into manageable, independent services and thinking about their interactions in a distributed fashion. It reinforces the idea that an "agent" is a specialized service within a larger system, leveraging the same foundational patterns.

Common Pitfalls & Troubleshooting in AI Architectures

Designing for AI introduces specific challenges on top of general distributed system complexities. Being aware of these can save significant time and effort.

1. **Over-engineering Agent Orchestration Too Early:** It's tempting to design a highly complex, dynamic agent orchestration layer from day one, anticipating every possible interaction. However, starting simpler (e.g., a sequential chain of agents) and only introducing more complex orchestration patterns (like state machines or dynamic task graphs) as needed can prevent premature complexity and unnecessary technical debt.
    - **Troubleshooting:** Start with a simple, linear flow. Introduce complexity only when a clear need arises from evolving requirements. Favor explicit control over implicit, dynamic routing until your system needs warrant it.
2. **Ignoring AI Model Lifecycle (MLOps Debt):** AI models are not static code. They need retraining, versioning, continuous evaluation, and secure deployment. Failing to integrate model deployment and monitoring into your infrastructure automation leads to stale models, unexpected performance degradation, or security vulnerabilities.
    - **Troubleshooting:** Establish MLOps (Machine Learning Operations) practices early. Implement automated pipelines for model validation, deployment, A/B testing, and rollback. Treat models as first-class citizens in your CI/CD process.
3. **Lack of Observability into Agent Reasoning:** While logging inputs and outputs is good, understanding why an agent made a particular decision or followed a specific path is crucial for debugging, improving, and auditing AI systems. Without this, your agents can become "black boxes."
    - **Troubleshooting:** Instrument agents to log intermediate steps, confidence scores, the "chain of thought," or the specific reasoning path taken. This is especially vital for explainable AI (XAI) and for identifying biases or errors.

4. **Underestimating Compute Costs for Inference:** Running large AI models, especially at scale with high query volumes, can be extremely expensive due to specialized hardware (GPUs) requirements. Without careful resource management and optimization, costs can spiral quickly.
    - **Troubleshooting:** Implement dynamic scaling for inference workers (scaling to zero when idle). Explore model quantization, distillation, or using smaller, more efficient models for less critical tasks. Leverage serverless inference platforms where appropriate to pay only for actual usage.

Summary

Congratulations! You've reached the end of our journey into modern systems engineering. This final chapter emphasized the enduring value of **systems thinking** – seeing the whole picture, understanding interconnectedness, and anticipating ripple effects. We've explored how to pragmatically **navigate architectural tradeoffs**, recognizing that every decision has consequences, and the best choice is always context-dependent.

Finally, we saw how these timeless engineering principles are directly applicable to the cutting edge of **AI and agentic workflows**. These intelligent systems, far from being magic, are built upon the very distributed system patterns you've mastered: microservices for decomposition, asynchronous communication for interaction, worker architectures for compute, and comprehensive observability for manageability.

Key Takeaways:

- **Systems Thinking is Paramount:** Always consider the entire system, not just individual components, to make effective architectural decisions. Understand how changes in one part affect the whole.
- **Tradeoffs are Inevitable:** Embrace the reality of tradeoffs (e.g., CAP theorem, cost vs. performance, complexity vs. flexibility) and learn to manage them strategically based on specific business needs and constraints.
- **AI/Agentic Systems are Distributed Systems:** They leverage microservices, asynchronous communication, worker pools, and robust observability just like any other scalable application.
- **Decomposition is Key for AI:** Break down complex AI tasks into smaller, specialized agents or services for better scalability, resilience, and maintainability.

- **Observability is Non-Negotiable:** For complex multi-agent systems, deep logging, metrics, and distributed tracing are critical for understanding agent behavior, debugging, and improving performance.
- **Timeless Principles Endure:** While technologies change rapidly, the core engineering principles of scalability, resilience, observability, and managing complexity remain constant and will serve you well throughout your career.

As you continue your journey, remember that architecture is a continuous process of learning, adapting, and making informed decisions. The principles you've learned here will serve you well, no matter how technology evolves.

References

- Microservices Architecture Style - Azure Architecture Center
- The CAP Theorem - IBM Cloud Learn
- Distributed Systems Design - Google Cloud
- What is MLOps? - Google Cloud
- Patterns for Distributed Systems - Martin Fowler

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.